بازگشت به کتابخانه C بدون فراخوانی تابع

(معماری x86)

چکیده

در این مقاله تکنیک های جدیدی را ارائه خواهیم کرد که بدون هر گونه فراخوانی تابع، امکان کارگر شدن حملات بازگشت به کتابخانه C (ret2libc) را روی فایل های اجرایی x86 ممکن می سازند. بردار حمله ی ما با ترکیب زیادی از توالی های کوتاه دستوری، گجت هایی (gadget) را می سازد که امکان انجام اعمال دلخواه را برای ما فراهم می آورند. با آنالیزهای ایستا (static) چگونگی کشف این توالی های دستوری را نشان خواهیم داد. این مقاله تحت مجموعه دستورات معماری x86 نوشته شده است.

۱. مقدمه

تکنیک های جدیدی را عرضه خواهیم کرد که امکان انجام حملات ret2libc را روی فایل های اجرایی x86 عملی می سازند که کارایی آن نیز برابر با عملیات تزریق کد معمول است. سپس اثبات خواهیم کرد که به دلیل امکان پذیر بودن حملات ret2libc روی W^X بسیار بیشتر از تصور ضعیف و بلااستفاده است (اگرچه که جلوی اجرای کدهای تزریق شده را می گیرد).

حملاتی که از تکنیک ارائه شده در این مقاله استفاده کنند به هیچ وجه تابعی را فراخوانی نمی کنند. در حقیقت از یک سری توالی دستوری در libc استفاده می کنند که ارتباطی به ترجمه شدن یا نشدن توسط اسمبلر ندارند. این مسئله بردار حمله ی ما را در قبال راه حل هایی تدافعی که توابع خاصی را از libc یا مولد کد (موتور) در اسمبلرها حذف می کنند، پایدارتر می سازد.

بر خلاف بردارهای حمله قدیمی، بردار ما از تعداد زیادی از توالی های کوچک دستورات استفاده می کند تا گجت های مورد نظر ما را بسازد. اجرا شدن این گجت ها، امکان اجرای اعمال دلخواه را به ما می دهند. طریقه ی ساختن این گجت ها را با استفاده از توالی های کوچکی که در توزیع خاصی از GLIBC می یابیم، نشان می دهیم. پس از آن به خاطر خاصیت و ماهیت مجموعه دستورات معماری x86 تخمین خواهیم زد که احتمالا در هر بلوک نسبتا بزرگ از کدهای قابل اجرای x86، توالی های دستوری وجود خواهند داشت که امکان ساختن گجت های مشابهی را به ما خواهد داد (این ادعا در حقیقت تز ما در این مقاله است). این مقاله به سه قسمت عمده تقسیم می شود:

1. **توصیف یک الگوریتم کارامد جهت آنالیز کردن libc. این الگوریتم برای بازیابی توالی های دستوری که می توانند در حمله مفید باشند، به کار می رود.**
2. **با استفاده از توالی هایی که از نسخه ی خاصی از GLIBC بازیابی کرده ایم (مورد بالا)، گجت هایی را پایه گذاری خواهیم کرد که امکان اجرای اعمال دلخواهمان را فراهم آورند؛ این روند به معرفی تکنیک های جدیدی می انجامد که شالوده ی اصلی چیزی خواهند شد که نام *برنامه نویسی بازگشت-گرا (return-oriented programming)* را برای آن انتخاب کرده ایم.**
3. **در حین انجام موارد فوق، شواهد محکمی را برای تز خودمان ارائه خواهیم کرد و بعلاوه قالبی را عرضه می کنیم که طریقه ی بررسی سیستمی های دیگر را نشان می دهد؛ تا به این ترتیب پشتیبانی و امکانات بیشتری را در آن سیستم ها بیابیم که در راستای بردار حمله ی ما مفید واقع شوند.**

بعلاوه این مقاله از چند بخش کوچک نیز تشکیل شده است. یک شلکد بازگشت-گرا را پیاده سازی کرده و شیوه ی استفاده از آنرا نشان می دهیم. سپس روی مبدا و منشا دستورات بازگشت (ret) در نسخه ی libc مورد استفاده مان مطالعه کرده و امکان حذف دستورات بازگشتِ غیرضروری را (بوسیله ی تغییر) در کامپایلر بررسی می کنیم. نهایتا ترسیم خواهیم کرد که تکنیک های حمله ی ما در حقیقت در دسته ی بزرگتری از حملات ret2libc جای می گیرند.

۱,۱ پیش زمینه: حملات و دفاعیات

نفوذگری را مجسم سازید که یک آسیب پذیری را در یک برنامه کشف کرده و قصد اکسپلویت کردن آنرا دارد. مفهوم اکسپلویت کردن در این مورد و با این زمینه (context) به معنی در دست گرفتن روند کنترل و اجرای برنامه توسط نفوذگر است تا بتواند اعمال مورد نظر خود را انجام دهد. می توان از سرریزهای مبتنی بر پشته، سرریزهای اعداد صحیح و آسیب پذیری های format string به عنوان آسیب پذیری های سنتی در این زمینه نام برد. در هر مورد نفوذگر باید دو عمل را به ثمر برساند: نخست اینکه راهی برای تغییر روند برنامه از روال عادی خود و نهایتا در دست گرفتن روند اجرای برنامه بیابد؛ دوم اینکه برنامه را وادار به اجرای اعمال مورد نظر خود کند.

در حملات سنتی واپاشی پشته، نفوذگر عمل اول را با جاینویسی آدرس بازگشت روی پشته به ثمر می رساند، بطوریکه آدرس بازگشت به جای اشاره به تابع فراخواننده (تابعی که فراخوانی را انجام داده است)، به کد مورد نظر نفوذگر اشاره کند [البته حتی اگر آدرس بازگشت، به تابع نیز اشاره داشته باشد، باز هم امکان استفاده از تکنیک های دیگر وجود دارد، مثل جاینویسی اشارگر قاب[1]]. عمل دوم را هم با تزریق کد به تصویر (image) بارگذاری شده ی پروسه انجام می دهد؛ آدرس تغییر یافته ی بازگشت به این کد اشاره خواهد داشت. به خاطر ماهیت توابع رشته ای در زبان C (که علت اصلی این نوع آسیب پذیری ها هستند)، کد تزریق شده باید عاری از بایت های پوچ (null) باشد. الیاس لوی (Aleph One) در مقاله ی معروفش طریقه ی کدنویسی برای معماری x86 در لینوکس را تحت این محدودیت ها تشریح می کند (که سبب اجرا شدن یک پوسته (shell) می شود)؛ اما شلکدها را می توان برای پلتفرم های مختلف و اهداف گوناگون نوشت.

این مقاله به ارزیابی میزان موثر بودن مکانیزم های امنیتی می پردازد که جهت جلوگیری از عمل دوم در بالا طراحی شده اند. مکانیزم های امنیتی گوناگونی وجود دارند که برای جلوگیری از اقدام اول طراحی شده اند (هر یک برای جلوگیری از گونه ی خاصی از حملات طراحی شده اند، مثل سرریزهای پشته، سرریزهای توده یا format string) که البته خارج از موضوع این مقاله هستند.

بیشترین تلاش متخصصان امنیت در منع نفوذگران از اجرای کدهای دلخواه، بیشتر منعطف بر جلوگیری از *اجرای کدهای تزریق شده* بوده است. اولین گونه از این مکانیزم ها، یعنی نرم افزار StackPatch، برای غیرقابل اجرا ساختن پشته، طرح حافظه (memory layout) را دستکاری می کرد. چون در حملات سرریز پشته شلکد معمولا در پشته تزریق می شود، این راهکار تا حدودی مناسب بود. مکانیزم کامل تر دیگری نیز بکار گرفته شد که با نام W^X معروف شد. این مکانیزم اطمینان حاصل می کند که هیچ مکانی از حافظه در تصویر یک پروسه، در آن واحد دارای هر دو خاصیت نوشتنی (writable) و اجرا کردنی (eXecutable) نباشد. با W^X هیچ محلی در حافظه وجود ندارد که نفوذگر امکان تزریق (نوشتن) و سپس اجرای کد را داشته باشد. پروژه ی PaX اصلاحیه ای را جهت پیاده کردن مکانیزم W^X برای لینوکس ارائه داد. محافظ های مشابهی در نسخه های اخیر OpenBSD نیز وجود دارند. بعلاوه دو شرکت AMD و Intel نیز بیت های NX (در AMD) و XD (در Intel) را در پردازنده های خود اضافه کرده اند. این بیت ها در هر صفحه (page) از حافظه، عمل اجرا کردن (execute) را غیر فعال می کنند و بدین ترتیب مکانیزم W^X را به طریق ساده تری پیاده می کنند. همچنین شرکت ماکروسافت نیز از نسخه ویندوز XP SP2 به بعد (پردازنده ی مقصد باید قابلیت NX/XD را پشتیبانی کند) از مکانیزم W^X پشتیبانی می کند.

حال که نمی توان کدی را تزریق کرد، در عوض می توان طبق نیاز از کدهایی استفاده کرد که از قبل در تصویر پروسه ی مورد حمله وجود دارند (Solar Designer برای اولین بار چنین راهبردی را پیشنهاد کرده بود). چون کتابخانه ی استاندارد C (libc) تقریبا در هر برنامه ی مبتنی بر یونیکس بارگذاری می شود، لذا یکی از محل هایی که می توان از کدهای آن استفاده کرد، libc است؛ به خصوص اینکه libc دارای روتین هایی است که بالقوه برای نفوذگر مفید واقع می شوند. به این ترتیب بود

[1] Frame pointer

که این حملات با نام ***Return-into-libc*** یا *بازگشت به کتابخانه C* معروف و شناخته شدند. اما در حقیقت می توان از هر کدی که در دسترس قرار دارد استفاده کرد، چه از سگمنت کد یک برنامه باشد، چه از کتابخانه ای که به آن برنامه متصل شده است. نفوذگر می تواند با مرتب کردن دقیق مقادیر روی پشته، سبب فراخوانی یک تابع دلخواه با آرگومان های دلخواه شود. در حقیقت سبب فراخوانی یک سری از توابع می شود، که یکی پس از دیگری فراخوانی می گردند.

۱,۲ نتیجه

شاید این سوال پیش آید که با وجود حملات ret2libc، استفاده از مکانیزم W^X چه ارزشی دارد. جواب به این ترتیب است که به دو دلیل حملات ret2libc نسبت به حملات تزریق کد دارای محدودیت بیشتری هستند:

۱. نفوذگر در یک حمله ret2libc می تواند یک تابع libc را پس از تابع دیگر فراخوانی کند، اما با این وجود فقط امکان اجرای کدهای *صریح (straight-line)* را دارد، بر خلاف حملات تزریق کد که در آنها می توان از دستورات انشعاب (branch) یا دیگر دستورات و اعمال دلخواه استفاده کرد؛
۲. نفوذگر فقط امکان فراخوانی توابعی را دارد که در سگمنت text برنامه و دیگر کتابخانه های بارگذاری شده در دسترس قرار دارد، لذا با حذف توابع مشخصی از libc می توان او را محدود کرد.

با در نظر گرفتن نتایج بالا گاها اینطور استنتاج می شود که استفاده از W^X منجر به تضعیف و محدود شدن نفوذگر می گردد. در این مقاله نشان خواهیم داد که این تصور اشتباه است. تکنیک های ret2libc جدیدی را ارائه خواهیم کرد که امکان اجرای اعمال دلخواه را فراهم می آورند (بنابراین محدودیت اجرای کدهای صریح را ندارند) و بعلاوه به فراخوانی هیچ تابعی نیاز ندارد، لذا حذف توابع از libc نیز موجب شکست این بردار حمله نمی شود.

۱,۲,۱ شالوده ی بردار حمله ی ما

شالوده ی حملات سنتی ret2libc توابع هستند و این توابع را می توان از libc حذف کرد. در نقطه مقابل شالوده ی حمله ی ما توالی های کوتاه کد هستند، که طول هر کدام دو یا سه دستور خواهد بود. بعضی از این دستورات در نتیجه ی استفاده از گزینه های مختلف تولید کد در زمان کامپایل، در libc قرار دارند. البته دسته ی دیگر این دستورات که در libc قرار دارند بدون دخالت کامپایلر است. حذف این توالی های کد بدون تغییرات اساسی در کامپایلر و اسمبلر در هر یک از این دو مورد بسیار دشوار است.

برای درک ماهیت توالی های دسته ی دوم در libc (یعنی توالی هایی که کامپایلر در آنها نقشی ندارد)، زبان انگلیسی را در نظر بگیرید. کلمات در زبان انگلیسی دارای طول متفاوتی هستند و هیچ جایگاه مشخصی در کاغذ نیست که یک کلمه ملزم به خاتمه یافتن در آن نقطه باشد و دیگری از آنجا شروع شود. کدهای اینتل معماری x86 نیز درست مثل زبان انگلیسی هستند که فاصله بین کلمات و علایم نشانه گذاری در آن وجود ندارد، لذا تمام کلمات در کنار یکدیگر به صورت یک کلمه ی بسیار بزرگ و واحد حضور دارند. پردازنده می داند که از کجا خواندن کلمه را شروع کند و تا چه میزان خواندن را ادامه دهد، در نتیجه قادر است که کلمات متفاوت را از آن رشته ی واحد و بزرگ بازیابی کرده و در نتیجه جمله ی مورد نظر را حاصل کند. در حین خواندن کلمات، فرد می تواند کلمات بیشتری را روی کاغذ بنویسد (کلماتی که از ابتدا روی کاغذ قرار نداشته اند)، یا از کلمات موجود کلمات جدیدی را استنباط کرده و بخواند. همه چیز بستگی به چگونگی خواندن فرد (چگونگی تفسیر پردازنده از دستورات) دارد. بعضی کلمه ها پسوند دیگر کلمات هستند، مثلا "dress" یک پسوند برای کلمه ی "address" است؛ بعضی دیگر حاوی حروف انتهایی یک کلمه و حروف ابتدایی کلمه ی بعدی هستند، مثلا "head" چنین حالتی را در عبارت "the address" دارد. حال یک مثال واقعی را برای معماری x86 نشان می دهیم که از کتابخانه ی استاندارد C ما در

محیط آزمایشگاهی به نام testbed اقتباس شده است (قسمت ۱,۲,۶ را ببینید). دو دستور در نقطه ی ورود ecb_crypt به صورت زیر رمزگذاری شده اند:

```
f7 c7 07 00 00 00        test $0x00000007, %edi
0f 95 45 c3              setnzb -61(%ebp)
```

با شروع از یک بایت جلوتر، نفوذگر چنین چیزی را خواهد داشت:

```
c7 07 00 00 00 0f        movl $0x0f000000, (%edi)
95                       xchg %ebp, %eax
45                       inc %ebp
c3                       ret
```

مقایس تناوب در اتفاق افتادن این موارد در وهله ی اول به خصوصیات زبان بر می گردد، که ما نام *هندسه* را بر آن اطلاق می کنیم. معماری مجموعه دستورات x86 بسیار چگال (dense) است، به این معنی که می توان یک جریان تصادفی از بایت ها را با ضریب احتمال بالا، به صورت مجموعه ای از دستورات صحیح و معتبر تفسیر کرد. بنابراین نه تنها پیدا کردن کلمات تصادفی (و در عین حال صحیح و معتبر) برای معماری x86 امکان پذیر است، بلکه در یک گام فراتر می توان حتی توالی هایی از کلمات را به این ترتیب یافت. شرط لازم جهت استفاده از یک *توالی* از دستورات در حمله ی ما، خاتمه یافتن آن توالی با دستور *بازگشت (return)* است که در هگزادسیمال معادل بایت **c3** است. بنابراین در آنالیز بلوک های بزرگ کد مثل libc انتظار اینکه به دستورات زیادی از این قسم برخوریم دور از انتظار نیست؛ کما اینکه این مسئله را در تز این مقاله نیز تدوین می کنیم.

تز ما در این مقاله: در هر بلوک از کدهای قابل اجرای x86 که به مقدار کافی بزرگ باشد، تعداد مناسبی از توالی های کد نیز برای نفوذگر وجود خواهد داشت که بوسیله ی آنها (و با استفاده از تکنیک ret2libc جدیدی که ما در این مقاله ارائه می کنیم) می تواند سبب انجام اعمال دلخواه در برنامه ی اکسپلویت شده گردد.

در مقابل در معماری هایی مثل MIPS که طول تمام دستورات ۳۲ بیت است و چیدمان (alignment) آنها نیز ۳۲-بیتی است، هیچ ابهامی راجع به ابتدا و انتهای یک دستور وجود نداشته و بعلاوه امکان تفسیر دستورات جدید از درون توالی های بایت فعلی نیز (به گونه ای که در بالاتر مثال زدیم) وجود ندارد. یک راه برای تضعیف بردار حمله ی ما استعمال چنین قابلیت هایی در معماری x86 است، مک کمنت و موریست در بخشی از طرح نرم افزار *ایزولاسیون نقص x86* [2] خود طرحی را برای چیدمان دستورات پیشنهاد داده اند که این کار را انجام می دهد. اما این طرح معایبی نیز دارد: نخست اینکه کدهایی که تحت چنین چیدمانی کامپایل می شوند، امکان فراخوانی توابعی که با این طرح کامپایل نگشته اند، ندارند. دوم اینکه عمل NOP Padding (پر کردن جاهای خالی با دستورات NOP) باعث می شود که کد واقعیِ کمتری در حافظه ی کش مربوط به دستورات (instruction cache) جای گیرد. به این ترتیب مثلا اجرای دستورات "andl $0xfffffff0, (%esp); ret" منجر به وابستگی داده ای (data dependency) در حافظه ی کش می شود که نهایتا به کاهش سرعت می انجامد. این کاهش سرعت در برنامه های معمولی و کلی، در مقایسه با روش های سنتی که سرعت بالایی دارند، غیر قابل پذیرش است. تاکید می کنیم که اگرچه روش های تدافعی از این قسم سبب تضعیف و وقفه در بردار حمله ی ما می شود، اما این الزاما به معنی جلوگیری کامل از این نوع حمله نیست.

در بخش دوم، چندین گام را معرفی کرده ایم که به منظور اجتناب از قرار دادن توالی های دستوری در درخت بکار می رود که کامپایلر سبب قرار گیری آنها شده است. اما نفوذگر تحت این الزام و محدودیت قرار ندارد و امکان وجود توالی های کافی و مناسبی نیز وجود دارد (که پسوندی از توابع در libc باشند) تا حمله به این ترتیب پایه ریزی شود.

[2] x86 Software Fault Isolation

مقاله ما که بر جزئیات مجموعه دستورات معماری x86 تکیه دارد از دو مقاله ی دیگر الهام گرفته است: یکی مقاله ی rix در مجله ی Phrack که چگونگی ساختن شلکدهای حرفی و عددی را در معماری x86 بررسی می کند و دیگری مقاله ای از Sovarel، Evans و Paul تحت عنوان "Where's the FEEB?" که طریقه ی غلبه بر چندین گونه از ایده های تصادفی سازی مجموعه دستور x86 را نشان می دهد.

۱,۲,۲ چطور توالی ها را پیدا کنیم

در بخش ۲، الگوریتم کارا و موثری را جهت آنالیز ایستای فایل های قابل اجرای x86 و کتابخانه ها معرفی می کنیم. در نسخه ی libc که ما در آزمایش ها استفاده می کردیم، این ابزار هزاران توالی را پیدا کرد و ما برای پایه ریزی حمله، مجموعه ی کوچکی از بین آنها را انتخاب کردیم. اخیرا مبحث آنالیز ایستا کاربرد بیشتری را به عنوان یک ابزار حمله پیدا کرده است. برای مثال Kruegel از یک مکانیزم پیچیده جهت اجرای مبتنی بر سمبل ها (symbolic execution) استفاده می کند. این مکانیزم به نفوذگر در یافتن روش هایی کمک می کند که بواسطه ی آنها بتوان کنترل برنامه را بعد از بازگشتن برنامه به حالت اولیه ی خود بدست گرفت. اینکار با هدف غلبه بر سیستم های تشخیص نفوذ مبتنی بر هوست (میزبان) انجام می شود. در جزئیات بردار حمله ی آنها (برخلاف بردار حمله ی ما)، نفوذگر می تواند کدهای "تزریق شده" ی دلخواه را اجرا کند. البته تکنیک های آنالیز ایستای آنها برای ما نیز مفید خواهد بود.

۱,۲,۳ چطور از توالی ها در یک حمله ی شناور استفاده کنیم

طریقه ای که ما در تکنیک return-oriented programming با libc فعل و انفعال برقرار می کنیم، با روشی که در حملات سنتی ret2libc بکار می رود، از سه جهت تفاوت دارد؛ و دقیقا به همین دلایل، ساختن گجت ها، امری ظریف و دشوار محسوب می شود. این سه تفاوت عبارتند از:

۱. توالی های کدی که فراخوانی می کنیم بسیار کوتاه خواهند بود– معمولا دو یا سه دستور – و زمانی که توسط پردازنده اجرا می شوند فقط جزو کوچکی از کار را برای ما انجام می دهند. اما در حملات سنتی ret2libc، شالوده ی اصلی حمله، توابع هستند که هر کدام اعمال قابل توجه و معنی داری انجام می دهند. در نتیجه حملات ما در یک سطح انتزاعی پائین تر انجام می پذیرند، مثل اسمبلر در مقایسه با یک زبان سطح بالا.
۲. توالی های کدی که فراخوانی می کنیم تقریبا هیچ کدام **prologue** و **epilogue** ندارند، و در حین حمله نیز مثل روش های استاندارد با یکدیگر زنجیر (chained) نمی شوند.
۳. توالی های کدی که فراخوانی می کنیم (که در حقیقت شالوده ی اصلی بردار حمله ی ما هستند) واسط های تصادفی دارند؛ در مقابل در حملات سنتی، واسط فراخوانی تابع به عنوان بخشی از ABI استانداردسازی شده است (البته تفاوت چهارمی را نیز بین توالی های کد ما و توابع libc به یاد داشته باشید: توالی های کدی که فراخوانی می کنیم به خودی خود توسط نویسندگان برنامه ها و توسعه دهندگان libc قرار داده نشده اند و بنابراین نمی توان براحتی آنها را حذف کرد).

در بخش ۳، طریقه ی ساختن گجت ها را نشان می دهیم و اینکه چطور بلوک های کوتاه موجود در پشته می توانند به زنجیر کردن چند توالی بیانجامند و در نتیجه ی آن، اعمال مورد نظر نفوذگر انجام شود. در این مقاله عمدتا گجت هایی را تشریح خواهیم کرد که اعمال ذخیره/بارگذاری، محاسباتی، منطقی، کنترل روند اجرا و فراخوانی های سیستمی را انجام می دهند.

تاکید می کنیم که اگرچه از توالی های مشخص کد در گجت ها استفاده می کنیم، اما می توانیم از دیگر توالی ها نیز استفاده کنیم؛ ضمنا اگرچه در پلتفرم های دیگر ممکن است توالی های مورد نظر ما در libc نباشند، اما قطعا توالی های مشابهی با

موارد مورد انتظار ما وجود دارند که می توان با همان ها نیز گجت هایی را مشابه گجت های خودمان ساخت (حداقل اگر تز ما درست واقع شود).

۱,۲,۴ کاربردهای قبلی توالی های کوتاه در حملات

چند حمله قدیمی ret2libc نیز از تکه های کوتاه کد در libc استفاده می کردند. تذکر این نکته لازم است که سگمنت های کدی که به فرم "pop %reg; ret" هستند و برای تنظیم ثبات ها مورد استفاده قرار می گیرند، در عین حال در معماری هایی که آرگومان ها از طریق ثبات بین توابع منتقل می شوند، وظیفه ی تنظیم آرگومان های تابع را نیز بر عهده دارند. معماری های SPARC و x86-64 از این دسته اند. از نمونه های دیگر می توان به توالی های "pop-ret" (کاری از nergal) و تکنیک پرش مبتنی بر ثبات ها یا "register spring" (معرفی شده توسط dark spyrit) اشاره کرد. تکنیک ما از منظر فراخوانی توابع libc با دیگر حملات تفاوت دارد. تکنیک های پیشین از توالی های کوتاه به عنوان یک عامل اساسی در زنجیر کردن فراخوانی توابع در libc یا اجرای کد تزریق شده و پرش به آن استفاده می کردند. اما در تکنیک ما این واقعیت نشان داده می شود که اگر توالی های کوتاه کد به درستی ترکیب شوند، تقریبا می توانند هر عمل دلخواه مورد نیاز نفوذگر را بدون فراخوانی هر گونه تابع انجام دهد. از تکنیک های پیشین، تکنیک Borrowed Code Chunks از Krahmer به تکنیک فعلی ما بیشترین تشابه را دارد. Krahmer از آنالیز ایستا برای یافتن توالی های register-pop استفاده می کرد. سپس ابزاری را برای تولید شلکد مطرح می کند که از این توالی ها استفاده کرده تا امکان انتقال آرگومان های دلخواه را به توابع libc فراهم سازد. به هر حال اکسپلویت هایی که با تکنیک Krahmer ساخته می شوند هنوز محدودیت کد صریح را دارند و متکی به توابع مشخصی در libc هستند (مثل دیگر حملات سنتی ret2libc).

۱,۲,۵ بایت های پوچ چه می شوند؟

یک خواننده ی دقیق احتمالا متوجه این نکته شده که بعضی از گجت های بحث شده در بخش ۳ نیاز به قرار گرفتن یک بایت پوچ (null) روی پشته دارند. یعنی چنین گجت هایی را نمی توان در payload یک سرریز پشته ی ساده استفاده کرد. این مسئله به دلایل زیر برای ما مشکلی ندارد:

۱. ما گجت ها خود را برای اجتناب از بایت های پوچ بهینه نکرده ایم. اگر بایت های پوچ مشکل ساز شوند می توان از همان تکنیک هایی که در تکنیک های استاندارد شلکدنویسی استفاده می شود، در جهت حذف آنها نیز بهره برد. برای مثال بارگذاری مقدار بلافصل صفر در EAX را می توان با توالی کد به فرم xor %eax, %eax; ret یا بارگذاری مقدار 0xffffffff به همراه یک دستور افزاینده جایگزین کرد. اگر آدرس توالی کد مورد نظرمان حاوی بایت پوچ باشد، می توانیم از نمونه های دیگری از آن توالی که آدرس آنها دارای بایت پوچ نیست استفاده کنیم، یا توالی متفاوتی را جایگزین کنیم.
۲. روش های دیگری وجود دارند که نفوذگر می تواند پشته را به طرق دیگری غیر از روش های استاندارد سرریزهای بافر جاینویسی کند. هیچ کدام از آنها محدودیت های یکسان ندارند. برای مثال در یک حمله format string مشکلی برای نوشتن بایت های پوچ در پشته وجود ندارد.
۳. ما از تکنیک خودمان نه تنها در یک محیط ایزوله، بلکه به عنوان روشی نوین در زمره ی حملات ret2libc استفاده می کنیم، که به همان اندازه در اکسپلویت ها کارا و موثر هستند.

رابطه ی مشابهی نیز در فعل و انفعال بین تکنیک ما و ASLR وجود دارد. می توان مستقیما از گجت هایی استفاده کرد که نیازی به آدرس های پشته نداشته باشند. بعلاوه می توان برخی از گجت هایی را که نیاز به دانستن آدرس های پشته دارند به طریقی جاینویسی کرد تا دیگر نیاز به آدرس ها نداشته باشند.

۱,۲,۶ محیط آزمایشگاهی libc

مثال های بررسی شده در این مقاله روی کتابخانه ی GLIBC توزیع شده در Fedora Core 4 انجام شده اند. نسخه ی آن libc-2.3.5.so بوده است. محیط آزمایشگاهی ما نیز یک کامپیوتر ۲,۴ گیگاهرتز با سیستم عامل Fedora Core 4 و هسته ی نسخه ی 2.6.14 بوده است.

۲. کشف توالی های دستوری مفید در libc

در این بخش الگوریتمی را برای کشف توالی های مفید کد در libc ارائه می دهیم. ما از توالی های خروجی از این الگوریتم استفاده می کنیم تا توالی های مناسب در گجت های مورد بحث در بخش ۳ را انتخاب کنیم. قبل از بررسی الگوریتم، باید تعریف دقیقتری از "توالی مفید دستوری" داشته باشیم. زمانی یک توالی دستوری را *مفید* می نامیم که بتوان آنرا در یکی از گجت ها استفاده کرد؛ شرط استفاده کردن یک توالی در گجت هم این است که اولا حاوی دستورات صحیح و معتبر باشد و دوما با دستور ret خاتمه یابد، و سوما هیچ دستوری تا زمان رسیدن به دستور ret در توالی نباشد که سبب تغییر روند اجرایی پردازنده شود (این دستور ret است که چنین نقشی را در عمل بازی می کند).

* الگوریتم:

create a node, *root*, representing the ret instruction;
place *root* in the trie;
for *pos* **from** 1 **to** *textseg len* **do**:
    if the byte at *pos* is ***c3***, i.e., a ret instruction, **then**:
        call BuildFrom(pos, root).

* تابع **BuildFrom**:

Procedure BuildFrom(index *pos*, instruction *parent_insn*):
    for *step* **from** 1 **to** *max_insn_len* **do**:
    if bytes [(*pos* − *step*) . . . (*pos* − *1*)] decode as a valid instruction *insn* **then**:
        ensure *insn* is in the trie as a child of *parent_insn*;
        if *insn* isn't boring **then**:
            call BuildFrom(*pos* − *step*, *insn*).

تصویر ۱: الگوریتمی که در این مقاله به کار می بریم

اگر دستورات موجود در یک توالیِ ایجاد شده در مرحله ی تولید کد در کامپایلر، معادل تابعی در libc باشد، آن توالی را *مناسب* یا *نامزد (intended)* می نامیم. طبق تزمان، الگوریتمی که توصیف کردیم تا جای امکان سعی در اجتناب از توالی های نامزد می کند، اگرچه استفاده از ret های نامزد در انتهای توالی ها مشکلی در بر ندارد.

دو مشاهده ما را در انتخاب ساختمان داده ی مناسب جهت ثبت یافته ها راهنمایی کرد. ابتدا اینکه هر پسوند برای یک توالی دستوری را می توان یک توالی دستوری مفید دانست. برای مثال اگر ما توالی "a; b; c; ret;" را در libc کشف کرده باشیم، آنگاه می توان برای توالی "b; c; ret;" نیز وجود خارجی قائل شد. دوم اینکه، تکرار و تناوب یک توالی برای ما اهمیتی ندارد، صرف یکبار وجود آن نیز برای ما کافی است. بر مبنای این مشاهدات تصمیم گرفتیم که توالی ها را در یک درخت پیشوندی ثبت کنیم. در ریشه ی این درخت، گره ای که نشانگر دستور ret است وجود دارد؛ ارتباط "فرزندی" در این درخت به این معنی است که دستور فرزند حداقل یکبار بلافاصله قبل از دستور والد در libc می آید. برای مثال اگر در یک درخت پیشوندی، گره pop %eax فرزندی از گره ریشه (که دستور ret است) باشد، آنگاه می توانیم اینطور نتیجه بگیریم که ما در یک جایی در libc، توالی pop %eax; ret را کشف کرده ایم. الگوریتم ما برای پر کردن گره های درخت پیشوندی از این واقعیت بهره می

گیرد که: "اسکن کردن رو به عقب دستورات با شروع از یک دستور ret، بسیار راحت تر از دیزاسمبل کردن رو به جلوی دستورات از هر محل ممکن (با این هدف که یک توالی دستوری خاتمه یافته به ret پیدا شود) است."

به هنگام اسکن کردن رو به عقب، دستورات اسکن شده و در نتیجه توالی حاصل شده تا به آن نقطه، پسوند تمام توالی هایی را که پیدا خواهیم کرد، شکل خواهد داد. سپس این توالی ها از نمونه های دستورات ret شروع می شوند که ما می توانیم libc را برای یافتن این دستورات اسکن کنیم. در پروسه ی بررسی رو به عقب از یک محل، همیشه باید سوالاتی از خودمان بپرسیم: آیا تک بایتی که قبل از توالی ما وجود دارد، نشانگر یک دستور تک بایتی صحیح و معتبر است؟ آیا دو بایت موجود قبل از توالی ما، نشانگر یک دستور دو بایتی صحیح و قانونی است؟ و این منوال را تا رسیدن به بزرگترین طول دستور در معماری x86 ادامه می دهیم.

اگر جواب هر کدام از سوالات فوق "مثبت" باشد، آنگاه یک توالی مفید در اختیار خواهیم داشت که آنرا می توان تا نقطه ی فعلی یک پسوند حساب کرد؛ و ما نیز برای بررسی های بیشتر باید آنرا به صورت بازگشتی به همان روش بررسی کنیم. به خاطر چگالی بالای معماری دستورات x86، ممکن است بیش از یکی از این سوالات جواب مثبت داشته باشند. تصویر ۱ الگوریتم ما را به صورت شبه-کد جهت یافتن توالی های مفید نشان می دهد.

۲,۱ دستورات مضر (boring)

تعریف دستورات مضر را به صورت زیر انجام می دهیم:

۱. دستور حاضر، یک دستور leave باشد و پس از آن نیز دستور ret آمده باشد؛ یا
۲. دستور حاضر، یک دستور pop %ebp باشد و بلافاصله پس از آن نیز دستور ret آمده باشد؛ یا
۳. دستور حاضر، یک دستور ret یا یک پرش غیرشرطی باشد.

آخرین دسته، جریان های دستوری ای هستند که کنترل برنامه را قبل از رسیدن به دستور ret به جای دیگری هدایت می کنند و بنابراین برای مقاصد مقاله ی ما مفید نیستند. اما دو دسته ی دیگر به ما امکان نادیده انگاشتن جریان های دستوری تولید شده توسط کامپایلر را می دهند. چون libc مورد بررسی ما با فعال بودن قابلیت frame pointer کامپایل شده بود، لذا توابع موجود در libc به احتمال بسیار زیاد با توالی "leave; ret" خاتمه می یابند (البته ممکن است با دستورات mov و pop معادلی برای آنها ساخته شود). ذکر این نکته مهم است که این شرایط، عملا توالی های دستوری مفید در قالب اکسپلویت های شناور (crafting) را از ما می گیرند. اما سه راه برای این کار هنوز وجود دارد. نخست اینکه، حتی اگر سعی در اجتناب از فراخوانی توابع حقیقی در libc داشته باشیم، ممکن است پسوندهای آن توابع مفید باشند؛ اگر این پسوندها کوتاه باشند، حذف کردن آنها برای نویسندگان کامپایلر دشوار خواهد بود. دوم اینکه همان خاصیتی که به ما امکان کشف دستورات غیرنامزد را می دهد، امکان کشف توالی های غیرنامزد (که به توالی "leave; ret" ختم می شوند) در بدنه ی توابع libc را نیز می دهد. سوم اینکه هر دو دستور leave و pop %ebp یک بایت طول دارند و این احتمال وجود دارد که توالی "leave; ret" که به آن می رسیم به هیچ وجه نامزد نباشد، اما در جریان بایتی libc (libc bytestream) مثل ret های غیرنامزد (در بخش ۵ توضیح داده شده) قرار گیرند. به خاطر داشته باشید که اگرچه تکنیک هایی که ما در تولید برنامه ها – که این برنامه ها خودشان از زنجیره ای از توالی های دستوری ساخته شده اند – توسعه می دهیم معمولا با دستورات leave سروکار ندارند، اما می توانیم با استفاده از روش های زنجیره کردن فریم ها (که در مقاله ی Nergal توضیح داده شده)، تکنیک هایمان را برای کارکرد با این شرایط نیز ارتقا دهیم. البته اینکه ما حتی بدون استفاده از تکه کدهایی که با شرایط بالا حذف می شوند همچنان امکان انجام حملات را داریم، شاهد دیگری است بر تز ما.

۲,۲ پیاده سازی و کارایی

پیاده سازی الگوریتم به شبه کدی انجامید که در بالاتر آمد. جهت کشف این مسئله که کدام قسمت از libc به صورت سگمنت قابل اجرا map می شود، کد ما باید هدرهای ELF فایل های libc را تفسیر (parse) نماید. از دو کتابخانه ی کمکی بهره می جوییم. یکی GNU LIBELF v0.8.9 برای تفسیر هدرهای ELF و دیگری libdisasm v0.2 با چند دستکاری محلی، جهت رمزگشایی دستورات x86.

آنالیز کردن ۱,۱۸۹,۵۰۱ بایت از سگمنت قابل اجرای libc به یک درخت پیشوندی با ۱۵,۱۲۱ گره می انجامد. اجرای این آنالیز روی یک کامپیوتر با پردازنده PowerPC G4 به قدرت ۱,۳ گیگاهرتز و حافظه ی ۱ گیگابایت حدود ۱,۶ ثانیه طول کشید. البته امکان ارتقای زمان اجرای الگوریتم وجود دارد – مثلا از رمزگشایی بایت های خاصی در libc که چندین بار تکرار می شوند اجتناب کنیم – اما به نظر ما همین مقدار از زمان نیز کافی است.

۳. برنامه نویسی بازگشت گرا

این قسمت را به عنوان کاتالوگی فرض کنید که اعمالی را که می توان با توالی های یافت شده در libc انجام داد، نشان می دهد. بعلاوه این قسمت به عنوان یک خودآموز کلی برنامه نویسی بازگشت گرا نیز عمل می کند. گجت ها یک واحد میانی سازمان یافته هستند. هر گجت مقادیر خاصی را روی پشته تعیین می کند که از یک یا چند توالی از دستورات libc استفاده می کنند. گجت ها اعمال مشخصی را انجام می دهند، مثل بارگذاری یک مقدار، XOR یا پرش.

برنامه نویسی بازگشت گرا از کنار هم قرار دادن گجت ها برای انجام اعمال مطلوب و مورد نظر تشکیل می شود. از لحاظ تست تورینگ این مجموعه ی گجت ها کامل هستند و در نتیجه برنامه های بازگشت گرا تقریبا می توانند هر کار ممکنی را با کدهای معماری x86 انجام دهند. تاکید می کنیم که توالی های کدی که گجت های ما به آنها اشاره دارند، به صورت ذاتی در libc قرار دارند؛ این توالی ها همچون خود گجت ها تزریق نشده اند. اگر بعضی از توالی های مورد استفاده ظاهر عجیبی دارند، به این دلیل است که می خواستیم بهترین توالی های موجود در testbed (کتابخانه استاندارد C ما در محیط آزمایشگاهی) را انتخاب کنیم.

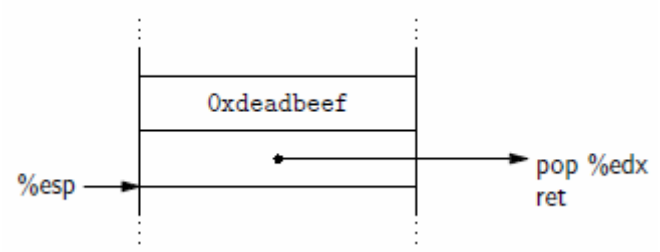
هر یک از گجت ها باید به طریق مشابهی وارد شوند: پردازنده یک دستور ret را با اشارگر پشته (که به کلمه ی زیرین گجت ها اشاره دارد) اجرا می کند. به این صورت، در یک اکسپلویت اولین گجت را باید طوری قرار داد که کلمه ی زیرین (انتهایی) آن، آدرس برگشت ذخیره شده ی یک تابع را روی پشته جاینویسی کند. گجت های بعدی را یا می توان بلافاصله بعد از اولین گجت قرار داد، یا با استفاده از گجت های کنترل جریان در بخش ۳,۳، آنها را در محل های دلخواه قرار داد.

۳,۱ بارگذاری/ذخیره

ما سه مورد را فرض می کنیم: بارگذاری یک مقدار ثابت در یک ثبات؛ بارگذاری محتویات یک محل حافظه در یک ثابت؛ و نوشتن محتویات یک ثبات در یک محل حافظه.

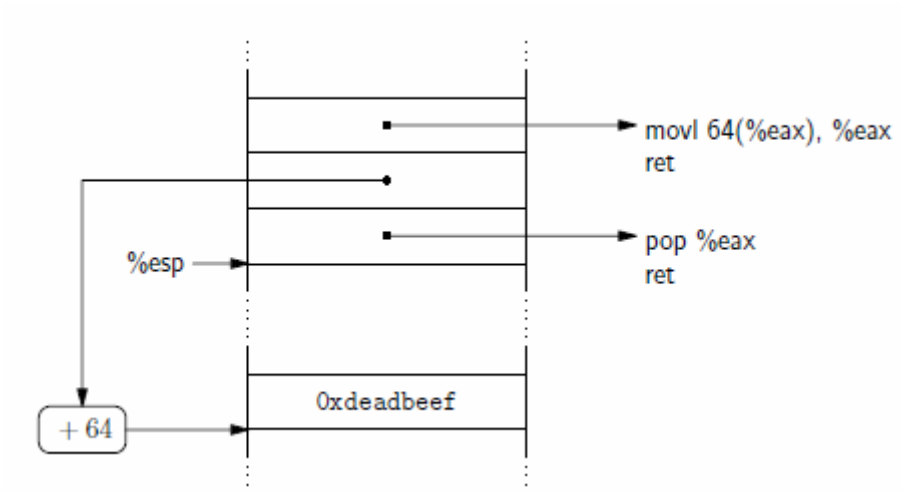
۳,۱,۱ بارگذاری یک مقدار ثابت

این خواسته را می توان با استفاده از یک توالی به فرم pop %reg; ret انجام داد. چنین مثالی را در تصویر ۲ تشریح کرده ایم.



تصویر ۲: بارگذاری مقدار ثابت 0xdeadbeef در ثبات EDX

در این تصویر و همچنین تصویرهای بعدی، اقلام موجود در نردبان نشانگر کلمات در پشته هستند؛ آنهایی که آدرس های بزرگتری دارند، در محل بالاتری در صفحه (page) قرار می گیرند. بعضی کلمات در پشته حاوی آدرس یک توالی از libc هستند. نشانه گذاری ما در این مورد به صورت یک اشارگر "از کلمه به توالی" است. کلمات دیگر حاوی اشارگرها به دیگر کلمات یا مقادیر بلافصل هستند. اینجا در مثال ما، زمانی که پردازنده توالی pop %edx; ret را اجرا می کند، دستور ret که سبب وارد شدن پردازنده به فاز اجرا کردن گجت می شود، خود سبب افزایش %esp به اندازه ی یک کلمه می شود؛ بنابراین دستور pop %edx کلمه ی بعدی (در اینجا، 0xdeadbeef) را از روی پشته در ثبات %edx بازیابی می کند. اینکار سبب افزایش دوباره ی ESP از انتهای گجت می شود بطوریکه دستور ret عملا باعث ادامه ی روند اجرا در گجت بعدی (که در بالای آن قرار گرفته) می شود.



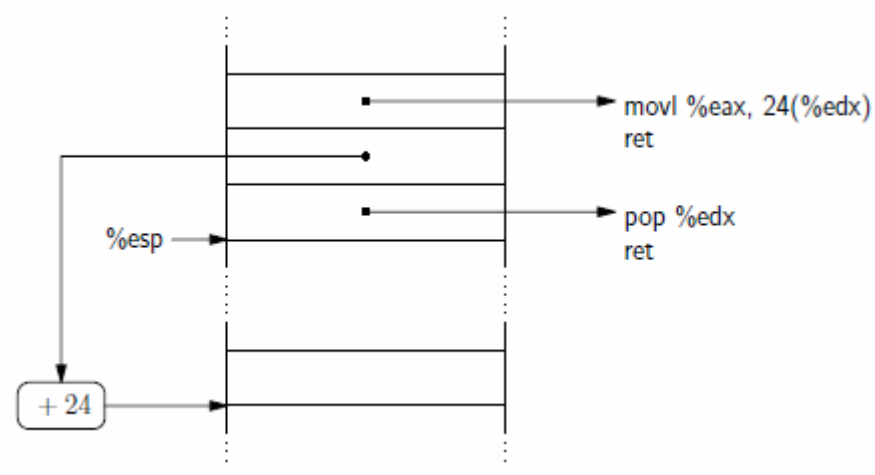
تصویر ۳: بارگذاری یک کلمه از حافظه در ثبات EAX

۳,۱,۲ بارگذاری از حافظه

می خواهیم با استفاده از توالی movl 64(%eax), %eax; ret مقداری را از حافظه در ثبات EAX بارگذاری کنیم. ابتدا آدرس را در EAX بارگذاری می کنیم. می توان اینکار را مثلا با رویه ی بارگذاری مقادیر ثابت (که در بالا توضیح داده شد) انجام داد. به خاطر آفست بلافصل در دستور movl، آدرس موجود در EAX در عمل باید ۶۴ بایت کمتر از آدرسی باشد که می خواهیم آنرا بارگذاری کنیم. پس از اینکه EAX حاوی محتویات محل مورد نظر حافظه بود، توالی movl را به کار می گیریم. جزئیات این رویه در تصویر ۳ نمایش یافته است. این نکته را خوب به خاطر بسپارید که "برای اشاره به سلول بعدی، باید به اشارگر سلول فعلی ۶۴ واحد اضافه کنیم".

۳,۱,۳ ذخیره در حافظه

ما از توالی movl %eax, 24(%edx); ret برای ذخیره ی محتویات ثبات EAX در حافظه استفاده می کنیم. با استفاده از رویه ی بارگذاری مقادیر ثابت (در بالاتر توضیح داده شد) آدرسی را که باید در EDX نوشته شود، بارگذاری می کنیم. جزئیات این رویه در تصویر ۴ به نمایش در آمده است.



تصویر ۴: ذخیره ی EAX در یک کلمه از حافظه

۳,۲ محاسبه و منطق

راهبردهای گوناگونی برای پیاده کردن اعمال محاسباتی و منطقی وجود دارند. راهبردی که ما انتخاب می کنیم و آنرا پارادایم واحد محاسبه و منطق (ALU) می نامیم به این صورت است: برای تمام اعمال، یکی از عملوندها EAX است و دیگری یک محل از حافظه. بسته به راحتی و بر حسب شرایط، یا EAX یا محل حافظه مقدار محاسبه شده را در خود ذخیره خواهد کرد. این راهبرد به ما امکان انجام اعمال حافظه-به-حافظه را به شیوه ی ساده ای می دهد: یکی از عملوندها را با استفاده از روش های بارگذاری-از-حافظه (بخش ۳,۱) در EAX بارگذاری می کنیم؛ عملگر را اعمال می کنیم؛ و اگر نتیجه هم اکنون در EAX نگهداری می شود، آنرا با استفاده از روش های نوشتن در حافظه (بخش ۳,۱) در حافظه قرار می دهیم.

در زیر بعضی از عمل ها را به همراه جزئیات توصیف می کنیم– به خصوص آنهایی که تکنیک های جدیدی برای ما محسوب خواهند شد – و بقیه را به صورت خلاصه تر مطرح می کنیم.

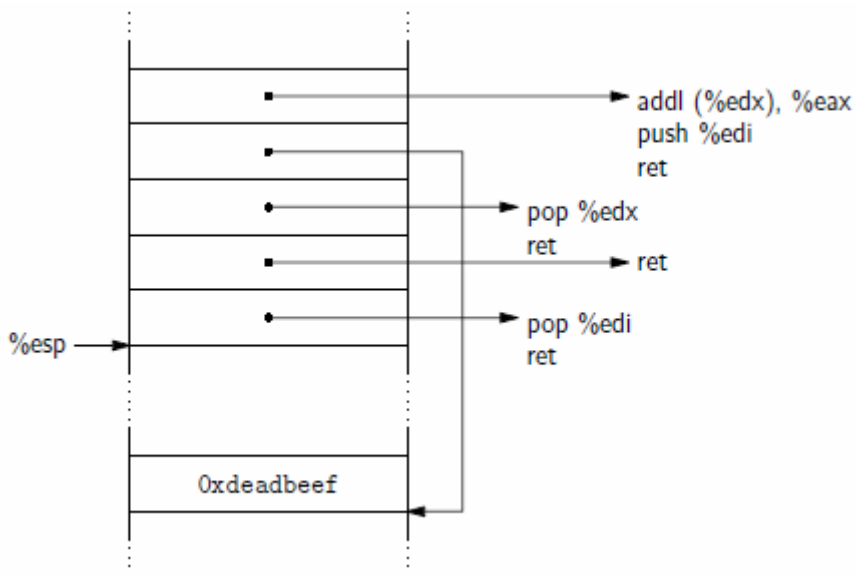
۳,۲,۱ جمع کردن

راحت ترین توالی برای انجام یک عمل جمع (که با پارادایم ALU ما نیز سازگار باشد) به صورت زیر است:

addl (%edx), %eax; push %edi; ret. **(1)**

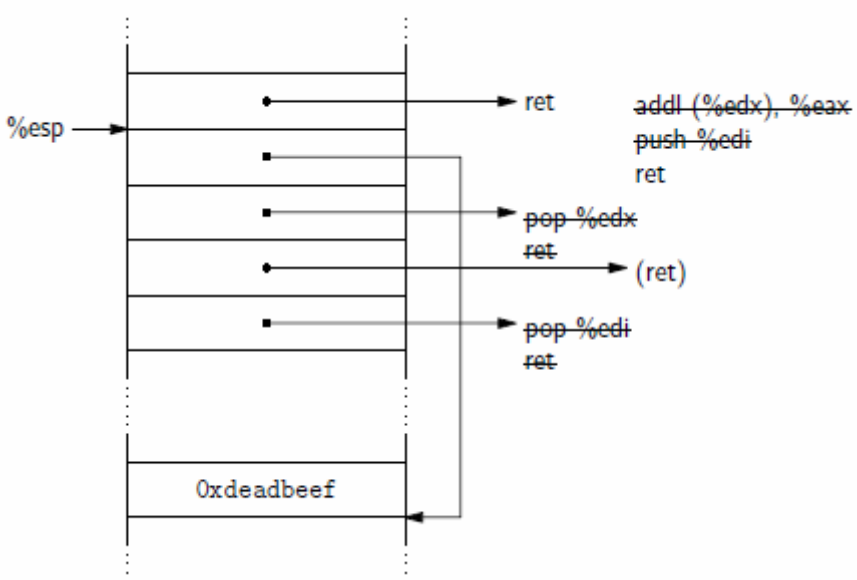
اولین دستور کلمه ی EDX را به EAX اضافه می کند که همان چیزی است که ما می خواهیم. اما دستور بعدی، مشکلاتی را بر سر راهمان قرار می دهد. اگرچه توالی pop-ret برای پیاده کردن عمل بارگذاری مقادیر ثابت مناسب بود، ولی توالی push-ret به دو دلیل مناسب نیست. یکی اینکه مقدار قرار گرفته در پشته، بلافاصله توسط دستور ret به عنوان آدرس توالی بعدی استفاده می شود. به این ترتیب روشن است که در push کردن مقادیر ما با محدودیت روبرو هستیم. دوم اینکه دستور push کلمه ای را در پشته جاینویسی می کند، بطوریکه اگر گجت را مجددا اجرا کنیم (مثلا در یک حلقه)، به طریقه ی قبلی رفتار نشان نخواهد داد.

ابتدا راهبرد ساده ای را ارائه می دهیم که مشکل دوم در آن به حساب نیامده است. قبل از استفاده از توالی دستور addl، آدرس یک دستور ret را در EDI قرار می دهیم. در برنامه نویسی بازگشت گرا، ret مثل NOP عمل می کند و به غیر از افزایش ESP، تاثیر دیگری ندارد. این چیدمان حمله در تصویر ۵ نشان داده شده است.



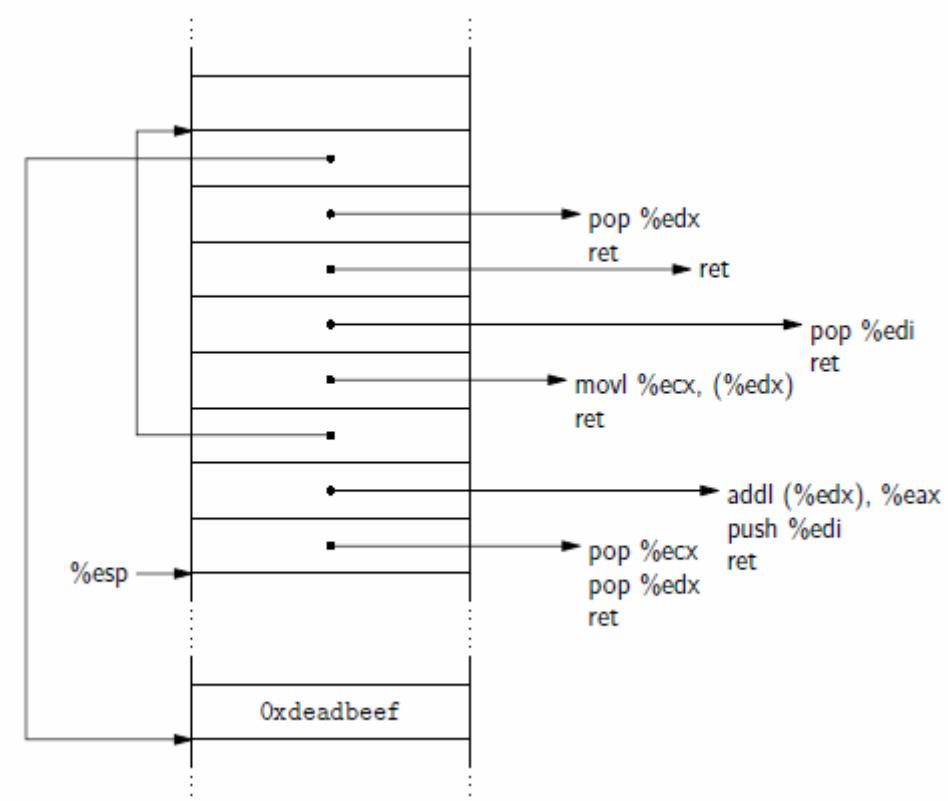
تصویر ۵: یک عمل جمع ساده و ذخیره ی آن در EAX

به این نکته توجه کنید که دستور push %edi باعث می شود که اولین کلمه (کلمه ی بالا) در پشته با محتویات EDI جاینویسی شود (یعنی به یک دستور ret اشاره کند). تصویر ۶ چیدمان حافظه را بلافاصله پس از اجرای دستور push %edi نشان می دهد.



تصویر ۶: شرایط یک عمل جمع ساده و ذخیره ی آن در EAX، پس از اجرای دستور push %edi

همان طور که می بینید، گجت با توالی push-ret تغییر یافته است و اجرای مجدد آن، عمل جمع را نتیجه نخواهد داد. اگر فقط نیاز به یک بار اجرا کردن گجت داشته باشیم، این مسئله مشکلی را در بر ندارد، اما اگر بخواهیم آنرا در یک روتین یا حلقه اجرا کنیم به مشکل بر می خوریم. راه حل ما در این موارد این است که آخرین کلمه در گجت را با آدرس (۱) به عنوان بخشی از کدِ گجت تنظیم کنیم. نمی توانیم از روش ذخیره در حافظه (بررسی شده در بخش ۳,۱) استفاده کنیم، چون محتوی EAX با یک عملگر جمع (add) اشغال شده است. در عوض از توالی دیگری استفاده می کنیم: movl %ecx, (%edx); ret. رویه ی کامل آنرا در تصویر ۷ ملاحظه کنید.



تصویر ۷: اجرای یک جمع تکرار شونده که نتایج آن در EAX ذخیره می شود

۳,۲,۲ دیگر اعمال محاسباتی

توالی neg %eax; ret به ما امکان محاسبه ی x در x را می دهد. اگر این توالی را با متدی که در بالا برای جمع کردن استفاده کردیم ترکیب کنیم، می توانیم مقادیر را از یکدیگر کم کنیم. در توالی هایی که در libc یافتیم، هیچ راه مناسب و راحتی برای ضرب وجود ندارد، اما می توان این عمل را با استفاده از جمع و اعمال منطقی (که در پائین بحث شده اند) شبیه سازی کرد.

۳,۲,۳ یای بولی (XOR)

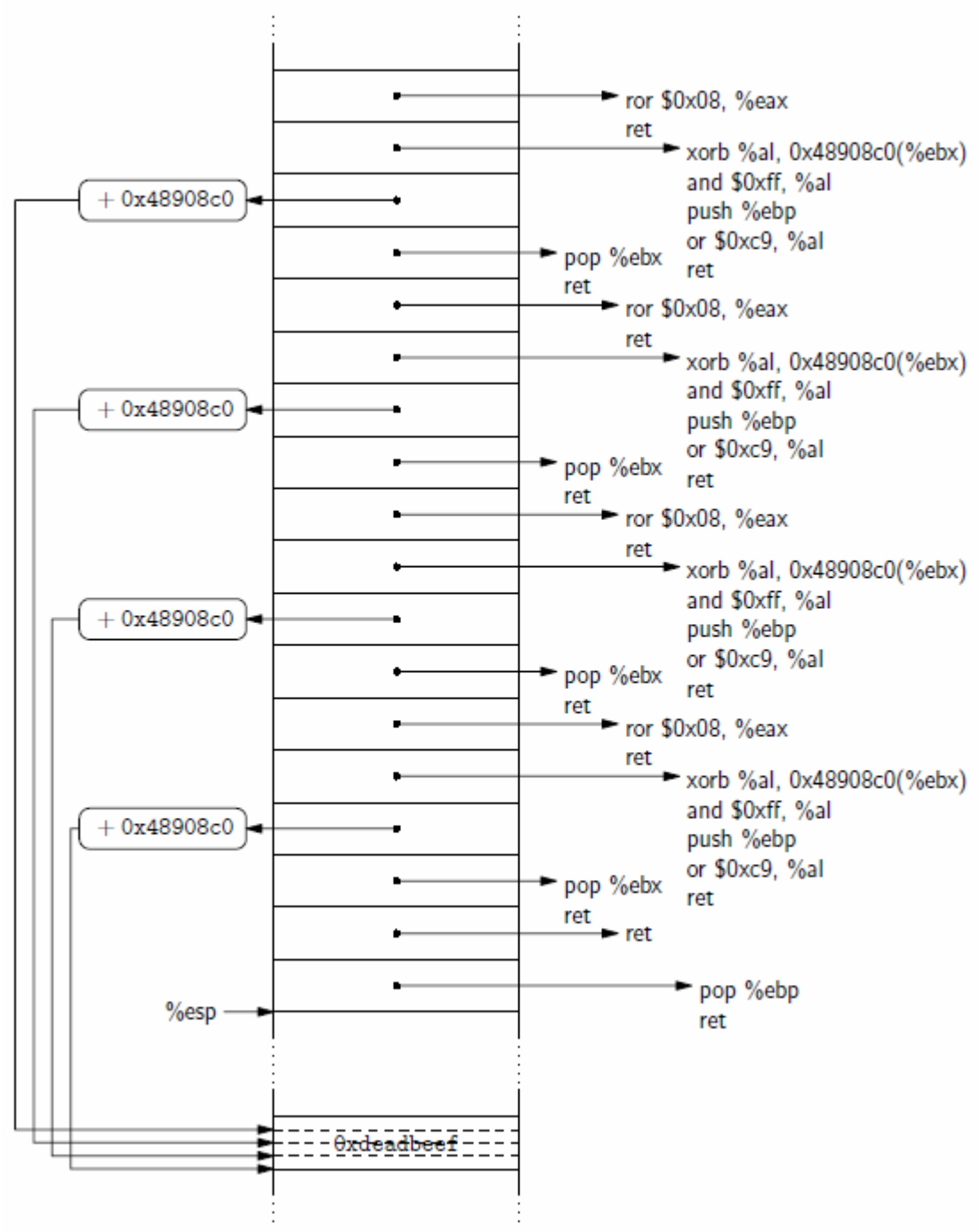
اگر یک توالی مثل xorl(%edx), %eax یا xorl %eax, (%edx) در اختیار داشتیم، می توانستیم یای بولی را درست مثل شیوه ی جمع کردن، پیاده سازی کنیم. اما چنین توالی هایی را در اختیار نداریم. البته به یک عملگر بایتی به فرم xorb %al, (%ebx) دسترسی داریم. اگر بتوانیم هر بایت از EAX را به ترتیب در AL حرکت دهیم، می توانیم یک عملگر کلمه ای XOR [۳] را با دو عملوند EAX و محل حافظه ی X داشته باشیم. این کار را با چهار بار تکرار کردن عمل انجام می دهیم که در آن EBX مقادیر x، x+1، x+2 و x+3 را خواهد گرفت. می توانیم براحتی با استفاده از توالی ROR $0x08, %eax; ret، عمل چرخش (rotate) را روی EAX انجام دهیم. اکنون تنها چیزی که باقی می ماند، فائق آمدن بر تاثیرات جانبی توالی XORB مورد استفاده مان است:

```
xorb %al, 0x48908c0(%ebx);        and $0xff, %al;
push %ebp;        or $0xc9, %al;    ret          (2)
```

آفست بلافصل در دستور XORB به این معنی است که مقادیری بارگذاری شده در EBX، باید به درستی چیده و تنظیم شده باشند. عملگرهای AND و OR باعث از بین رفتن مقدار در AL می شوند، اما قبلا در کنار آن از AL استفاده کرده ایم،

[3] منظور از عملگر "کلمه ایِ XOR"، انجام عمل XOR بر روی داده ای به طول یک کلمه است.

بنابراین مشکلی وجود ندارد (اگر بخواهیم عمل دیگری را با مقدار موجود در EAX انجام دهیم، باید آنرا مجددا از حافظه بارگذاری کنیم). دستور push به این معنی است که باید آدرس دستور ret را در EBP بارگذاری کنیم؛ همچنین اگر می خواهیم دستور XOR تکرار پذیر باشد، باید دستورات XORB را هر بار در گجت جاینویسی کنیم (به همان شیوه ای که برای جمع های تکرار پذیر در بالاتر توضیح دادیم). تصویر ۸ جزئیات یک عملگر XOR (عملگر پیشین) را نشان می دهد.



تصویر ۸: انجام XOR از ثبات EAX

۳,۲,۴ اعمال AND، OR و NOT

اعمال AND و OR بیتی را می توان با عملگرهای بایتی نیز به راحتی، مشابه با رویه ای که در بالا برای XOR توصیف شد، پیاده سازی کرد. توالی های کد به ترتیب به صورت

```
andb %al, 0x5d5e0cc4(%ebx); ret
```

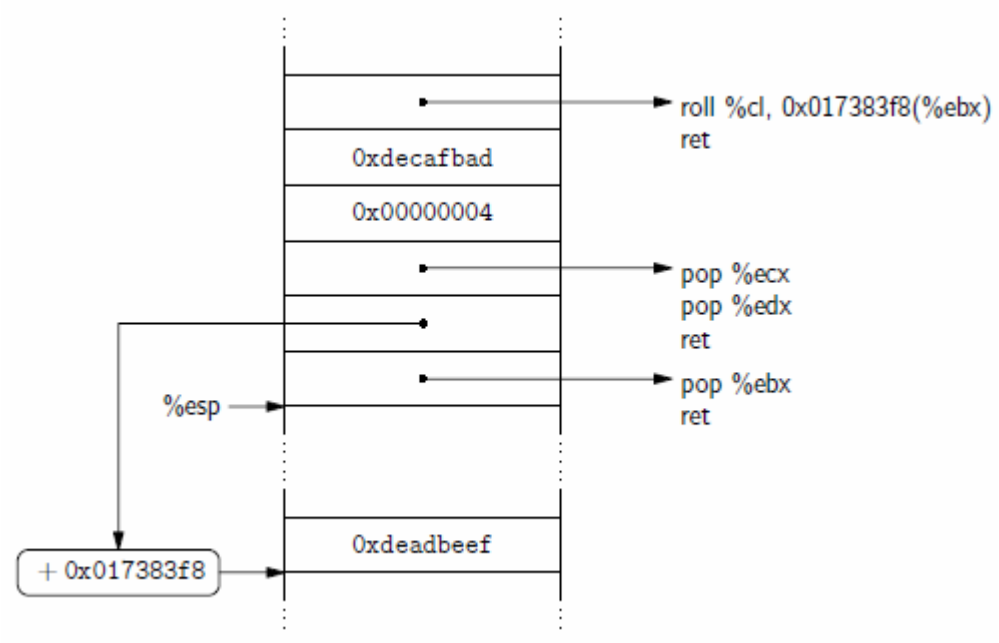
و

```
orb %al, 0x40e4602(%ebx); ret.
```

هستند. این توالی های کد نسبت به (۲)، تاثیرات جانبی کمتری برای XOR دارند بنابراین استفاده از آنها آسان تر است. عمل NOT بیتی را نیز می توان با XOR کردن مقدار با بیت های ۱ (روشن) پیاده سازی کرد.

۳,۲,۵ شیفت ها و چرخش ها

ابتدا شیفت ها و چرخش ها را با یک مقدار بلافصل (ثابت) فرض می کنیم. در این مورد به جای پیاده کردن تمام مجموعه ی شیفت ها و چرخش ها، یک عملگر واحد را پیاده سازی می کنیم: یک چرخش به چپ، که ساختن دیگر اعمال با استفاده از آنها ساده تر می باشد: یک چرخش به راست به اندازه ی k بیت، معادل یک چرخش به چپ به اندازه ی 32-k بیت است؛ یک شیفت به اندازه k بیت به هر جهت، معادل یک چرخش به اندازه ی k بیت به همراه یک ماسک از بیت هایی است که می خواهیم صفر باشند؛ که مورد اخیر را می توان خود با استفاده از روش AND بیتی که در بالاتر توضیح داده شد، محاسبه کرد. توالی کدی که برای چرخش استفاده می کنیم roll %cl, 0x17383f8(%ebx); ret است. گجت متناظر را در تصویر ۹ با جزئیات بیشتر مشاهده کنید.



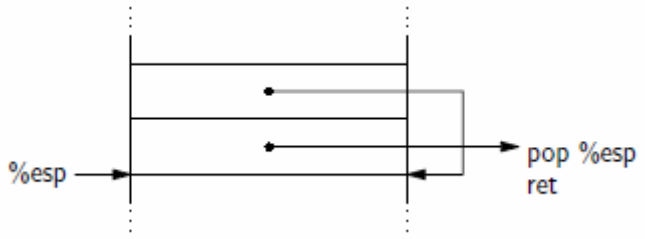
تصویر ۹: انجام عمل چرخش بلافصل با چهار بیت به سمت چپ، روی یک کلمه از حافظه

اکنون شیفت ها و چرخش ها را به صورت تعداد متغیری از بیت ها در نظر می گیریم. گجت موجود در تصویر ۹ مقدار ECX را از پشته می خواند. اگر می خواهیم این مقدار به محل دیگری از حافظه بستگی داشته باشد، می توانیم بسادگی آنرا از حافظه خوانده و در محلی از حافظه که ECX از آنجا خوانده می شود، بنویسیم. پیاده سازی شیفت ها با بیت های متغیر مقداری دشوارتر است، چون در آنصورت باید به ماسک متناظر با بیت های شیفت کننده دسترسی داشته باشیم. ساده ترین راه برای رسیدن به این مقصود ذخیره کردن یک جدول ارجاعی ۳۲ کلمه ای از ماسک ها در برنامه است.

۳,۳ کنترل جریان

۳,۳,۱ پرش غیرشرطی

چون در برنامه نویسی بازگشت گرا، اشارگر پشته جای اشارگر دستور را در کنترل روند اجرا می گیرد، یک پرش غیرشرطی نیاز به تغییر مقدار ESP دارد تا به گجت جدید اشاره داشته باشد. این کار را می توان به سادگی با توالی دستوری pop %esp; ret انجام داد. تصویر ۱۰ گجتی را نشان می دهد که با پرش رو به عقب به خودش، یک حلقه ی نامتناهی می سازد. حلقه ها قبلا نیز در اکسپلویت های ret2libc مورد توجه قرار داشته اند (چالش Esoteric#2 را از Gera ببینید).



تصویر ۱۰: ساختن یک حلقه ی بینهایت بوسیله ی یک پرش غیرشرطی

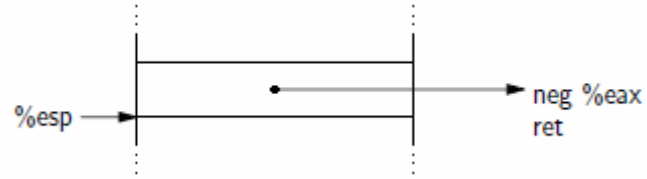
۳,۳,۲ پرش های شرطی

این پرش ها با حقه ی بیشتری همراه هستند. در اینجا روشی را برای نیل به پرش های شرطی توسعه می دهیم. برای شروع ابتدا مسائلی را مرور کنیم. دستور cmp عملوندهای خود را مقایسه کرده و بر اساس رابطه ی آنها، مجموعه ای از فلگ ها را در ثباتی به نام EFLAGS تنظیم می کند. در برنامه نویسی معماری x86، معمولا استفاده ی مستقیم از cmp لازم نیست، چون در بسیاری از دستورات، فلگ ها خودشان به عنوان تاثیر جانبی آن دستور تنظیم می شوند.

دستورات پرش شرطی (Jcc) در صورت وجود شرایط خاصی در فلگ های تنظیم شده، عمل می کنند. چون این پرش به عنوان تغییر در اشارگر دستور انگاشته می شود، دستورات پرش شرطی برای برنامه نویسی بازگشت گرا مفید نیستند؛ چون چیزی که ما نیاز داریم یک تغییر شرطی در اشارگر پشته است (نه اشارگر دستور). بنابراین استراتژی مورد نظر ما به ترتیب در سه مرحله به هدف می رسد:

۱. انجام عملی که سبب تنظیم یا پاک شدن فلگ های مورد نظر ما شود.
۲. انتقال فلگ ها از EFLAGS به یک ثبات چند-منظوره و محافظت و ایزوله کردن از فلگی که مد نظر ما می باشد.
۳. استفاده از فلگ مورد نظرمان برای اخلال در ESP تا پرشی به میزان مورد نظرمان را تداعی نماید.

برای اولین مورد، بنابه دلایلی که ذیلا بیان می گردد از فلگ نقلی (carry) استفاده می کنیم. با استفاده ی منحصر از این فلگ، می توان به مکملی از اعمال مقایسه ای استاندارد دست یافت. به طریق ساده تر نیز می توانیم با اِعمال neg بر آن، صفر بودن یا نبودن آنرا مشخص کنیم. دستور neg (و گونه های دیگر آن) مکمل-دوی عملوند را حساب می کند و به عنوان تاثیر جانبی نیز اگر مقدار صفر باشد، فلگ CF را تنظیم و در غیر این فلگ را پاک می کند. تصویر ۱۱ ساده ترین مورد را نشان می دهد که در آن مقدار مورد نظر ما برای بررسی در EAX قرار دارد (به خاطر داشته باشید که این مسئله به خاطر سازگاری با پارادایم ALU ما در بخش ۳,۲ است).



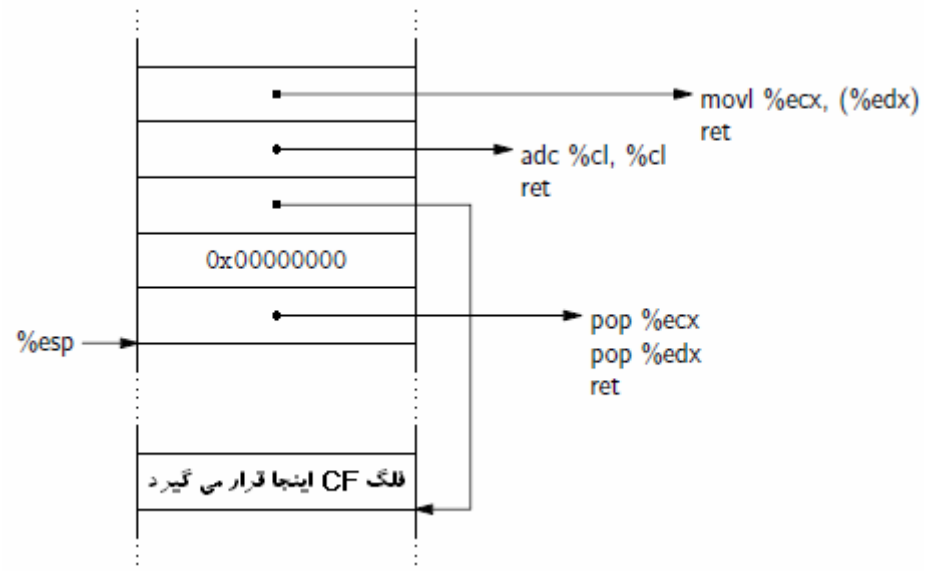
تصویر ۱۱: پرش های شرطی، فاز اول: اگر EAX به صفر رسید، CF را غیر فعال کن؛ اگر EAX غیرصفر است، CF را فعال کن.

اگر بخواهیم مساوی بودن دو مقدار را بررسی کنیم، می توانیم یکی را از دیگری کم کنیم، و سپس طبق آنچه که در بالا توضیح داده شد با دستور neg صفر بودن مقدار حاصله را بررسی کنیم. اگر بخواهیم بزرگ بودن یک مقدار را از دیگری بررسی کنیم، می توانیم مجددا مقدار اولی را از دومی کم کنیم؛ دستور sub (و گونه های مختلف آن) در صورت بزرگتر بودن مفروق از مفروق منه، فلگ CF را فعال می کند.

برای مورد دوم، راه طبیعی و معمول استفاده از دستور lahf است که پنج فلگ محاسباتی (SF، ZF، AF، PF و CF) را در AH ذخیره می کند. متاسفانه این دستور در توالی های libc موجود نبود که از آن استفاده کنیم. راه دیگر استفاده از دستور pushf است که کلمه ای را روی پشته قرار می دهد که شامل تمام محتویات EFLAGS است. این دستور در دسترس ما قرار دارد، اما برای استفاده از آن در برنامه نویسی بازگشت گرا، باید همچون دیگر دستورات push-ret حقه هایی به کار بریم.

در عوض ما راه سومی را انتخاب می کنیم. چندین دستور از فلگ نقلی به عنوان ورودی استفاده می کنند؛ علی الخصوص چرخش های چپ و راست با رقم نقلی (یعنی RCL و RCR) و همچنین جمع با رقم نقلی (ADC).

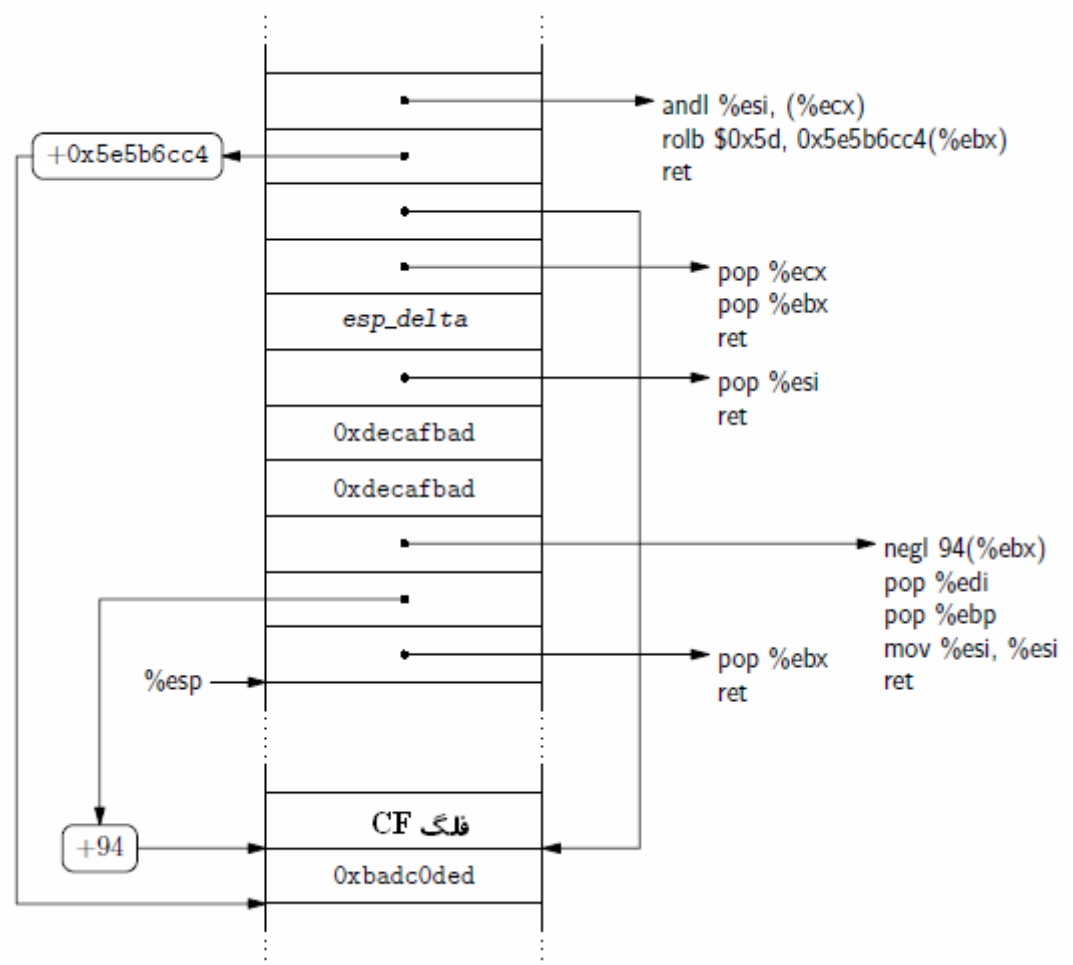
جمع با رقم نقلی، جمع دو عملوند خود را به همراه فلگ نقلی محاسبه می کند، که در جمع های چند کلمه ای مفید است. اگر دو عملوند این دستور را صفر اختیار کنیم، نتیجه ی جمع، بسته به فعال بودن فلگ نقلی، یکی از دو مقدار صفر یا یک خواهد شد (یعنی همان چیزی که ما می خواهیم). این روند را می توانیم به سادگی انجام دهیم. ابتدا ثبات ECX را صفر می کنیم و سپس از توالی adc %cl, %cl; ret استفاده می کنیم. جزئیات این رویه در تصویر ۱۲ نشان داده شده است. به خاطر داشته باشید که می توانیم با جمع کردن مقادیر CF در خلال چند تست و بررسی و ترکیب آنها با عملگرهای منطقی (که در بخش ۳,۲ توضیح داده شد)، عبارت های پیچیده ی بولی را ارزیابی کنیم.



تصویر ۱۲: پرش های شرطی، فاز دوم: بسته به اینکه فلگ CF فعال یا غیرفعا ل است، مقدار "صفر" یا "یک" را در کلمه ای از حافظه که با عنوان "فلگ CF اینجا قرار می گیرد" مشخص شده است، قرار بده.

برای سومین مورد، ما یک کلمه در حافظه داریم که مقدارش صفر یا یک است. آنرا به صورتی تغییر می دهیم که حاوی صفر یا دلتای ESP باشد؛ که در آن دلتای ESP مقداری است که در صورت صحیح بودن شرط می خواهیم ESP را به آن اندازه تغییر داده یا اصطلاحا در آن *انحراف (perturbation)* ایجاد کنیم. یک راه برای این کار را مثال می زنیم. در مکمل-دوی عدد ۱، تمام بیت ها یک (all-1) هستند و در مکمل-دوی عدد صفر نیز تمام بیت ها صفر (all-0) هستند، بنابراین negl را بر کلمه ی حاوی CF اعمال می کنیم که بیت های آن تماما صفر یا یک است. سپس عملگر AND بیتی را روی مقدار حاصله اعمال می کنیم. دلتای ESP حاوی کلمه ای خواهد بود که مقدارش یا دلتای ESP است یا صفر. جزئیات این روند در تصویر ۱۳ آمده

است (توالی های دستوری مورد استفاده ی ما، دارای تاثیرات جانبی هستند که باید روی آنها کار کرد، اما رویه بـه خـودی خـود سر راست و بدون مشکل است).

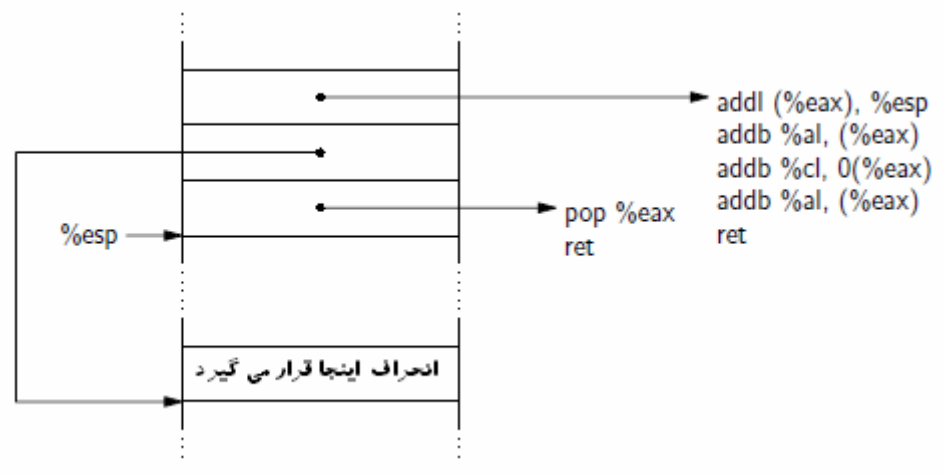


تصویر ۱۳: پرش های شرطی، فاز سوم–قسمت اول: کلمه ای (که با عنوان "فلگ CF" مشخص شده) از حافظه که حاوی صفر یا یک است را به یکی از دو مقدار esp_delta یا صفر تبدیل کن.

اکنون انحراف مطلوب را در اختیارداریم. با استفاده از توالی زیر مـی تـوانیم آنـرا روی اشـارگر پـشته اعمـال کنـیم، کـه در آن EAX نشانگر مقدار اختلاف مکان (displacement) می باشد.

```
addl (%eax), %esp; addb %al, (%eax);
addb %cl, 0(%eax); addb %al, (%eax); ret
```

دستورات دیگر در این توالی، وظیفه ی از بین بردن اختلاف مکان را بر عهده دارند. اما همان طور که قبلا نیز آنرا بکـار بـردیم، مشکلی بوجود نخواهد آمد. جزئیات این رویه را در تصویر ۱۴ مشاهده کنید.



تصویر ۱۴: پرش های شرطی، فاز سوم-قسمت دوم: انحراف را به مقدار کلمه ای که با عنوان "انحراف اینجا قرار می گیرد" مشخص شده است، بر روی اشارگر پشته اعمال کن. انحراف نسبت به انتهای گجت نسبی است.

۳,۴ فراخوانی های سیستمی

مشاهدات ما نشان می دهد که فراخوانی های سیستمی، پوشش های (wrapper) ساده ای در libc دارند که اغلب آنها به صورت زیر رفتار می کنند:

۱. آرگومان ها را از پشته به ثبات ها منتقل می کنند، و شماره ی فراخوانی سیستمی را در EAX قرار می دهند؛
۲. کنترل هسته را در دست می گیرند (البته این کار به صورت *غیر مستقیم* و با کتابخانه ای انجام می شود که خود هسته آنرا ارائه داده است، یعنی linux-gate.so.1)؛
۳. سیستم را برای وجود خطا بررسی می کند و بر آن اساس مقدار برگشتی مناسب را در EAX تنظیم می کند.

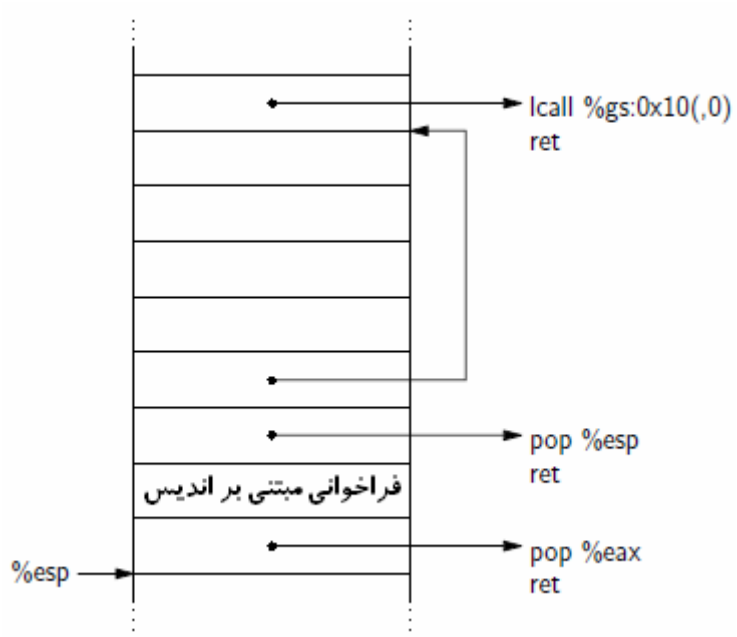
کد نوشته شده ی زیر برای umask، نمونه ای از این دست است:

```
89 da                    mov %ebx, %edx
8b 5c 24 04              movl 4(%esp), %ebx
b8 3c 00 00 00           mov $0x0000003C, %eax
65 ff 15 10 00 00 00     lcall %gs:0x10(,0)
89 d3                    mov %edx, %ebx
c3 ret
```

تابع lcall در سگمنت GS، __kernel_vsyscall را فراخوانی می کند. این تابع دستور sysenter یا int 0x80 را صادر می کند. اگر پارامترهای فراخوانی سیستمی را خودمان تنظیم کرده و به تابع پوششی (wrapper) در گام دوم بلافاصله قبل از lcall در خط چهارم پرش کنیم (برای umask)، می توانیم هر فراخوانی سیستمی را با هر پارامتری احضار و فراخوانی کنیم.

برای فراخوانی های سیستمی، به جای اینکه از توالی های غیرنامزد و نسبتا طولانی در جای دیگری استفاده کنیم، می توانیم از توالی های نامزد در libc استفاده کنیم. چون تقریبا تمام برنامه های مفید از فراخوانی های سیستمی استفاده می کنند، لذا این پیش نیاز که تابع پوششی یک فراخوانی سیستمی در اختیار ما باشد، بسیار منطقی تر و با مسمی تر از پیش نیاز در دسترس بودن روتین های خاص، مثل system است. در لینوکس، می توانیم با فراخوانی مستقیم __kernel_vsyscall خود را از این فرضیات برهانیم، البته بعد از اینکه با تفسیر (parse) بردارهای کمکی ELF آنرا پیدا کردیم.

در تصویر ۱۵ جزئیات یک گجت را نشان می دهیم که یک فراخوانی سیستمی را انجام می دهد. آرگومان ها را می توان در ثبات ها قرار داد که به ترتیب در EBX، ECX، EDX، ESI، EDI و EBP قرار می گیرند.



تصویر ۱۵: یک فراخوانی سیستمی بدون آرگومان. اندیس (شماره) فراخوانی سیستمی در دومین کلمه از پائین ذخیره شده است. آرگومان ها را می توان از قبل در ثبات های EBX، ECX و غیره بارگذاری کرد. چون تابع vsyscall مقادیری را روی پشته قرار می دهد، فضایی برای آن کنار می گذاریم (توابع مبتنی بر sysenter نیز اینطور هستند). به خاطر داشته باشید که کلمه ای که با lcall اشاره می کند نیز باید جاینویسی شود؛ و هر بار برای بازیابی آن نیاز به یک نسخه ی قابل تکرار (repeat) از این گجت می باشد.

اکنون راهی برای بارگذاری این ثبات ها با مقادیر مناسب نشان می دهیم. با استفاده از تکنیک های موجود در بخش ۳،۱، مقدار مطلوب را در EAX بارگذاری می کنیم، سپس آنرا در کلمه ی دوم یک گجت می نویسیم، بطوریکه کلمه ی اول آن گجت به یک توالی کد به فرم pop %r32; ret اشاره داشته باشد (که در آن r32 ثباتی است که باید تنظیم گردد).

۳،۵ فراخوانی های تابعی

عاقبت به این نتیجه می رسیم که هیچ چیز نمی تواند ما را از فراخوانی توابع دلخواهمان در libc باز دارد. این مسئله در حقیقت پایه ی اکسپلویت های قبلی و سنتی ret2libc است و تکنیک های ملزوم را می توانید در مقاله ی Nergal دنبال کنید؛ بحث راجع به تکنیک frame faking [۴] از موارد مورد علاقه ی ماست. ذکر یک نکته را در این زمینه لازم می دانیم: بهترین راه این است که توابع را در حالتی فراخوانی کنیم که اشارگر پشته در آنها به قسمتی از پشته اشاره کنند که آن قسمت ها توسط دیگر کدهای بازگشت-گرا مورد استفاده نباشند، چون ممکن است آن توابع در حین استفاده از پشته سبب آسیب زدن به گجتی شوند که بعدا می خواهیم مجددا فراخوانی کنیم.

۴. شلکد بازگشت-گرا

اکنون به عنوان یک مورد کاربردی از تکنیک های مطرح شده در بخش ۳، یک شلکد بازگشت گرا را ارائه می دهیم. شلکد تابع execve را جهت اجرای یک پوسته فراخوانی می کند. پیش نیازهای این مسئله عبارتند از:

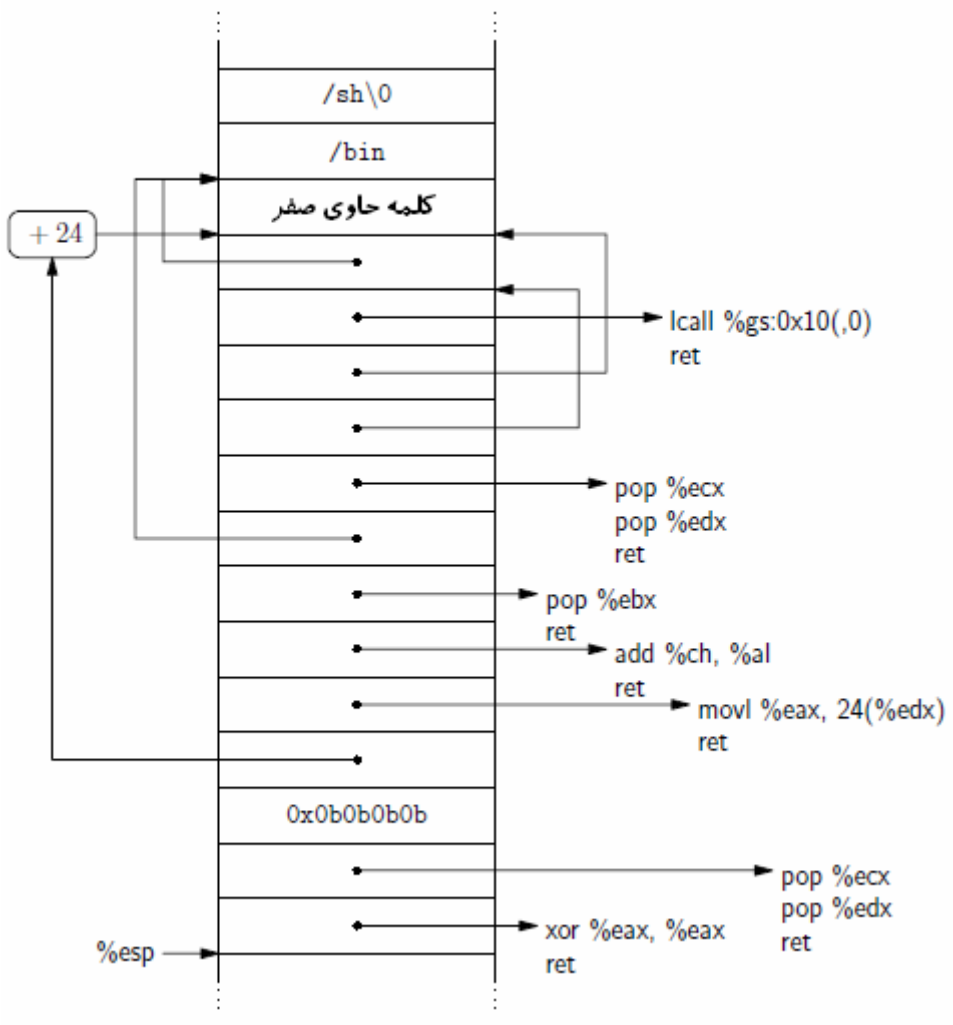
۱. تنظیم شماره ی فراخوانی سیستمی (یعنی 0xb) در EAX،
۲. تنظیم مسیر برنامه جهت اجرا (یعنی رشته ی "/bin/sh") در ثبات EBX،

[4] رویه ای که در آن قاب های تقلبی در پشته می سازند

۳. تنظیم بردار آرگومانی argv در ثبات ECX به آرایه ای متشکل از دو اشارگر؛ که اولی به رشته ی "/bin/sh" اشاره دارد و دومی نیز NULL است.

۴. تنظیم بردار محیطی envp در ثبات ECX به آرایه ای از یک اشارگر که NULL باشد.

جزئیات شلکد در تصویر ۱۶ قابل مشاهده است.



تصویر ۱۶: شلکد

ما رشته ی "/bin/sh" را در دو کلمه ی ابتداییِ شلکد ذخیره می کنیم؛ سپس از دو کلمه ی بعدی برای آرایه ی argv استفاده می کنیم و از کلمات بالاتر نیز مجددا برای آرایه ی envp استفاده می کنیم. می توانیم اشارگرهای متناسب را به صورت بخشی از خود شلکد تنظیم کنیم، اما برای اجتناب از بایت های پوچ باید اشارگر پوچ را پس از تزریق شدن شلکد صفر کنیم. ادامه ی شلکد به طریق زیر رفتار می کند:

- کلمه ی ۱ (از پائین): ثبات EAX را صفر می کند.
- کلمات ۲-۴: آدرس کلمه ی دوم در argv منهای ۲۴ (بخش ۳,۱,۲ را ببینید) را در EDX قرار می دهد، و برای آماده شدن جهت تنظیم اندیس راخوانی تابع، کلمه ای حاصل از بیت های صفر (all-0) را در ECX قرار می دهد.
- کلمه ۵: دومین کلمه در argv را به صفر تنظیم می کند.

- کلمه ۶: با دستکاری کم ارزش ترین بایت (یعنی AL) در ثبات EAX، مقدار این ثبات (EAX) را برابر با 0x0b می کند.
- کلمه ۷-۸: ثبات EBX را به رشته ی "/bin/sh" اشاره می دهد.
- کلمه ۹-۱۱: آدرس آرایه ی argv را در ثبات ECX، و آدرس آرایه ی envp را در ثبات EDX تنظیم می کند.
- کلمه ۱۲: روند اجرای کرنل را در دست می گیرد (قسمت ۳,۴ را ببینید).

با این نظر که محل هایی که آدرس های توالی های دستوری libc و آدرس های پشته به آن اشاره می دارند، هیچ کدام حاوی بایت های پوچ (null) نیست، این شلکد نیز به غیر از خاتمه دهنده ی رشته ی "/bin/sh" حاوی هیچ شلکدی نیست. می توان با واداشتن شلکد به ساختن این آدرس ها در زمان اجرا بوسیله ی بررسی ESP و کار کردن روی آن، بر مسئله ی وجود بایت های پوچ در آدرس های پشته فائق آمد؛ این کار امکان قرار گرفتن شلکد در محل های مختلف پشته را (بدون احتیاج به دست بردن در شلکد برای تغییر در آدرس ها و در نتیجه در محل) نیز به ما می دهد.

می توان با استفاده از تکنیک های بخش ۱,۲,۵ با بایت های پوچ در آدرس های libc فائق آمد. فرض کنید در یک برنامه، libc در آدرس پایه ی 0x03000000 بارگذاری شده است، و این برنامه تابعی دارد که علیه حملات سرریز بافر آسیب پذیر است و همچنین آدرس بازگشت این تابع در آدرس 0x04fffefc قرار گرفته است. در این مورد شلکد فوق نتیجه ی زیر را در بر خواهد داشت:

```
3e 78 03 03 07 7f  02 03 0b 0b 0b 0b 18 ff   ff   4f
30 7f  02 03 4f  37 05 03 bd ad 06 03 34 ff   ff   4f
07 7f  02 03 2c  ff  ff   4f  30  ff   ff 4f  55 d7 08 03
34 ff   ff   4f  ad fb  ca de 2f  62 69 6e 2f  73 68 00
```

به خاطر داشته باشید که به غیر از آخرین بایت، هیچ بایت پوچی وجود ندارد. همچون دیگر مثال های ارائه شده از کدهای بازگشت گرا در این مقاله، این شلکد نیز تنها از کدهای استفاده می کند که از قبل در libc وجود داشته باشند و حتی با وجود W^X نیز عملکرد داشته باشند. در ضمیمه ی الف، نشان می دهیم که در حقیقت می توان این شلکد را بوسیله ی یک سرریز بافر روی پشته برای اکسپلویت کردن یک برنامه ی آسیب پذیر بکار برد.

۵. فهرستی آماری از دستورات ret

در این بخش آماری را راجع به ماهیت بایت های c3 در سگمنت قابل اجرای libc نشان می دهیم. روش ما به این صورت است که برای هر بایت c3 که می یابیم، این مهم را چک می کنیم که آیا آن بایت در محدوده ی بایت های متعلق به یک تابع استخراج (export) شده در بخش SYMTAB از libc هست یا خیر. اگر چنین بود، آنرا در آمارمان به حساب می آوریم. سپس تابع را دیزاسمبل می کنیم تا زمانی که بیابیم کدام دستور حاوی بایت c3 است. همه ی سگمنت قابل اجرای libc توسط توابع استخراجی (exported) پوشش نیافته است. قسمتی از سگمنت توسط هدرهای ELF اشغال شده است و قسمتی دیگر نیز توسط توابع ایستا که در بخش SYMTAB نامگذاری نشده اند. با این حال این روش مشاهدات مناسبی را جهت نتیجه گیری نهایی به ما ارائه می دهد.

از ۹۷۵,۶۲۶ بایت بررسی شده، ۵,۴۸۳ تا از آنها c3 بودند، یعنی یکی در هر ۱۷۸ بایت (این مقدار بیشتر از مقداری است که انتظار داشتیم، یعنی یکی در هر ۲۵۶ تا).

- دقیقا ۳,۴۲۹ دستور ret وجود دارد. چون فقط ۳,۱۵۷ نقطه ی ورود (entry point) یکتا در بخش SYMTAB لیست شده است، می توان نتیجه گرفت که بعضی توابع بیشتر از یک دستور ret دارند.
- ۱,۶۹۲ از این مقدار در بایت ModR/M برای یک دستور add imm32, %ebx (با آپکد 81 **c3** imm32) وجود دارند. جمع بلافصل بخشی از گروه بلافصل ۱ است (آپکدهای ۸۰ تا ۸۳) که در آنها بیت های ۳ الی ۵ از بایت

ModR/M برای رمزگذاری یک پسوند آپکد استفاده می شود. در این مورد بیت های ۶ و ۷ (۱۱)، ثبات بودن یا نبودن مقصد را مشخص می کنند؛ بیت های صفر تا ۲ (۰۱۱) ثبات EBX را مشخص می کنند، و بیت های ۳ تا ۵ (۰۰۰) یک دستور جمع را مشخص می کنند. برای مقایسه 81 c2 دستور add imm32, %edx را رمزگذاری خواهد کرد و 81 cb دستور or imm32, %ebx را رمزگذاری می کند. جداول ۱-۲ و A-4 از [12] را ببینید.

- ۲۹۰ مورد مربوط به جابجایی های بلافصل بوده اند. از این تعداد، ۲۸۳ مورد مربوط به آفست های دستورات (۱۰۹ فراخوانی نسبی)، ۱۰۰ مورد مربوط به پرش های شرطی نسبی و ۶۴ مورد مربوط به پرش های غیرشرطی نسبی می باشد. ۱۷ مورد باقیمانده نیز آفست های داده ای را مشخص می سازند، مثل movb %al, -61(%ebp) که آپکد آن 88 45 c3 می شود.
- ۳۵ مورد در بایت درست و صحیح ModR/M قرار می گیرند که در آن %eax و %ebx به ترتیب به عنوان منبع و مقصد معلوم می شوند. از این تعداد ۳۳ تای آنها مربوط به دستور add %eax, %ebx (با آپکد 89 c3) می شود. و دو مورد دیگر دستورات shrd %cl, %eax, %ebx و shld %cl, %eax, %ebx است که آپکدهای آنها به ترتیب 0f ad c3 و 0f a5 c3 هستند.
- ۲۸ مورد مربوط به ثابت های بلافصل در دستورات add، mov و movw است.
- ۹ مورد در بایت SIB اتفاق می افتد که آدرس دهی را به فرم (%ebx, %eax, 8) مشخص می کنند. همه ی این موارد نمونه هایی از دستور mol r/m32, r32 هستند که در آن بایت ModR/M آدرس دهی را به صورت SIB+disp32 مشخص می کند (آپکدهای آنها 8b modr/m c3 imm32 است که modr/m در آن به فرم 10bbb100 است).
- ۱ مورد در دستور نقطه اعشاری fld %st(3) اتفاق می افتد که آپکد آن d9 c3 است (یا به صورت کلی تر، به فرم زیر میباشد: (d9 c0+iencodes fld %st(i)).

۵.۱ آیا می توان از ret های جعلی اجتناب ورزید؟

برخی تغییرات جزئی در GCC ممکن است به تولید شدن libc بدون بایت های غیرنامزد c3 بیانجامد. برای مثال، هر رویه می تواند فقط یک نقطه ی خروج داشته باشد (با توالی استاندارد leave; ret) که نقاط خروج قبلی بتوانند به آن پرش کنند. در اعمال جمع می توان ثبات EBX را از نقش انباشتگر باز داشت. از انتقال داده از EAX به EBX نیز می توان خودداری کرد یا اینکه از دستورات دیگری غیر از mov برای این انتقال استفاده کرد. وضعیت قرار گیری دستورات را نیز می توان در بسیاری موارد فشرده کرد تا از آفست های با بایت c3 اجتناب دوری جست.

ممکن است یک چنین استراتژی ای در حذف بایت های c3 غیرنامزد از فایل های اجرایی تولید شده موفقیت آمیز باشد. هزینه این استراتژی، کامپایلری است که شفافیت (transparency) کمتر و پیچیدگی بیشتری دارد. بعلاوه باعث افت کارایی چشم گیری نیز در استفاده از ثبات ها، در معماری هایی خواهد شد که ثبات های کمی در اختیار دارند، مثلا EBX معمولا به عنوان یک انباشتگر استفاده می شود، چون بر اساس قواعد فراخوانی (calling convention) اینتل، بر خلاف EAX، ECX و EDX، مقدار این ثبات توسط تابع فراخوانده شده روی پشته ذخیره می شود.

البته باید این نکته را نیز روشن کنیم که اگرچه رویه ی بالا ممکن است سبب حذف ret های غیرنامزد شود، اما سبب حذف توالی های دستوری غیرنامزد که با ret خاتمه می یابند، نمی شود. چون با وجود اینکه نفوذگر به انتخاب رشته هایی محدود می شود که پسوندی از توابع قانونی libc باشند، اما همچون گذشته هنوز نیاز نیست که این رشته ها از ابتدای یک محدوده ی دستوری نامزد انتخاب کند. برای مثال نقطه ی ورود تابع svc_getreg از libc با کد زیر خاتمه می یابد:

```
81 c4 88 00 00 00        add $0x00000088, %esp
5f                       pop %edi
```

```
5d                      pop %ebp
c3                      ret
```

با جداکردن چهار بایت آخر از این قطعه، نتیجه ی زیر را خواهیم داشت:

```
00 5f 5d                addb %bl, 93(%edi)
c3                      ret
```

البته یک مشکل بنیادین وجود دارد. گجت های توصیف شده در بخش ۳ فقط از توالی های دستوری استفاده می کردند که به بایت های c3 ختم می شد، که برای ما کافی بود. اما معماری مجموعه دستور x86 در عمل دارای چهار آپکد است که نقش یک دستور بازگشت (return) را ایفا می کنند: ***c3*** (بازگشت نزدیک)، ***c2 imm16*** (بازگشت نزدیک همراه با تمیز کردن پشته)، ***cb*** (بازگشت دور) و ***ca imm16*** (بازگشت دور همراه با تمیز کردن پشته). گونه هایی که همراه با تمیز کردن پشته انجام می شوند، آدرس بازگشت را از پشته بازیابی (pop) می کنند، اشارگر پشته را به اندازه ی imm16 بایت افزایش می دهند؛ این مورد در قواعد فراخوانی که در آنها پاک کردن آرگومان از پشته بر عهده ی تابع فراخوانده شده است، مفید واقع می شود. بازگشت های دور، مقدار ثبات CS و EIP را از پشته بازیابی می کنند. استفاده از این گونه ها در اکسپلویت هایی که توصیف کردیم دشوارتر است. برای بازگشت های دور، سگمنت صحیح کد نیز باید در پشته قرار گیرد؛ برای بازگشت هایی که پشته را نیز تمیز می کنند، باید از وقوع Stack Underflow ها نیز اجتناب کرد. با این حال استفاده از این گونه ها امکان پذیر است، و پاک کردن نمونه های همه ی این چهار نوع نیز مشکل می باشد. بعلاوه اگر امکان بارگذاری مقادیر بلافصل در ثبات ها را داشته باشیم (مثل استفاده از تکنیک هایی که در بخش ۳,۱ توصیف شد)، آنگاه می توانیم از توالی هایی استفاده کنیم که به ret ختم نشوند. برای مثال اگر EBX به یک دستور ret در libc اشاره داشته باشد، آنگاه هر توالی خاتمه یافته با jmp %ebx را می توان استفاده کرد. این در واقع پرش های مبتنی بر ثبات ها با زمینه ی ret2libc است. با کمی بررسی بیشتر به این نتیجه می رسیم که اگر کلمه ی موجود در imm(%esp) حاوی آدرس یک دستور ret باشد، آنگاه می توانیم از توالی هایی استفاده کنیم که به بایت jmp imm(%esp) خاتمه می یابند و این منوال برای بقیه ی ثبات ها نیز امکان پذیر است.

در نهایت ذکر این نکته نیز لازم است که تصویر قابل اجرای libc حاوی ناحیه هایی است که برای کدهای قابل اجرا در نظر گرفته نشده اند، هدرهای ELF یکی از مهم ترین موارد از این دست است. این هدرها ممکن است حای دستورات بازگشت نیز باشند که دستکاری در کامپایلر نمی تواند در وجود آنها تاثیری داشته باشد.

۶. نتیجه گیری و کارهای آتی

در این مقاله روش جدیدی را برای طراحی و مدیریت اکسپلویت های ret2libc روی معماری x86 ارائه دادیم. این روش کاملا با روش های قبلی تفاوت دارد. با استفاده از آنالیز ایستا توالی های کوتاه دستوری را یافتیم؛ سپس طریقه ی ترکیب این توالی ها را در قالب گجت ها نشان دادیم که امکان اجرای اعمال دلخواه را برای نفوذگر فراهم می آورند. موارد گوناگونی برای تحقیقات آینده وجود دارند. اولین مورد گام برداشتن در جهت اتوماتیک سازی و یکپارچه شدن با تکنولوژی های موجود است. با در اختیار داشتن مجموعه ای از گجت ها، فرد می تواند یک موتور زیربنایی بازگشت گرا برای gcc یا llvm بسازد. برای ساختن گجت ها از خروجی الگوریتم مان استفاده کردیم. البته این امکان نیز وجود دارد تا توالی های کد موجود را با هدف چگونگی ترکیب آنها جهت ساختن گجت ها، به صورت اتوماتیک آنالیز کرد.

دومین مورد بررسی صحت اعتبار یا نقض کردن اعتبار تز ما روی دیگر پلتفرم هاست. اگرچه گجت هایی که توصیف کردیم همگی از توزیع خاصی از GNU LIBC اقتباس شده اند، اما تکنیک هایی که جهت کشف توالی ها و ترکیب آنها در قالب گجت ها ارائه کردیم برای همه ی پلتفرم ها عمومی است. آنالیز مقدماتی از msvcrt.dll نویدبخش صحت ادعای بود.

ضمیمه الف. جزئیات شلکد مربوط به برنامه آسیب پذیر

کد مقصد موجود در تصویر ۱۷ را در نظر بگیرید. به خاطر سرریز، آدرس بازگشت به آدرس 0x04ffffefc تنظیم شده و buf نیز دارای آدرس 0x04ffffeb8 است- یعنی ۶۸ بایت پائین تر از آدرس بازگشت (۶۴ بایت برای بافر و ۴ بایت هم اشارگر قاب ذخیره شده). روند اجرا شدن کدها در این مورد به صورت زیر خواهد بود:

```
% ./target ‘perl -e ’print "A"x68,
        pack("c*",0x3e,0x78,0x03,0x03,0x07,0x7f,0x02,0x03,
                0x0b,0x0b,0x0b,0x0b,0x18,0xff,0xff,0x4f,
                0x30,0x7f,0x02,0x03,0x4f,0x37,0x05,0x03,
                0xbd,0xad,0x06,0x03,0x34,0xff,0xff,0x4f,
                0x07,0x7f,0x02,0x03,0x2c,0xff,0xff,0x4f,
                0x30,0xff,0xff,0x4f,0x55,0xd7,0x08,0x03,
                0x34,0xff,0xff,0x4f,0xad,0xfb,0xca,0xde,
                0x2f,0x62,0x69,0x6e,0x2f,0x73,0x68,0x0)’‘

sh-3.1$
 void do_map_libc(void) {
        int fd;
        struct stat sb;
        fd = open("libc-2.3.5.so", O_RDONLY, 0);
        fstat(fd, &sb);
        mmap((void *)0x03000000, sb.st_size,
                PROT_READ|PROT_EXEC, MAP_FIXED|MAP_SHARED, fd, 0);
 }

 void do_map_stack(void) {
        int fd;
        fd = open("/dev/zero", O_RDONLY, 0)
        mmap(0x04f000000, 0x001000000, PROT_READ|PROT_WRITE,
        MAP_FIXED|MAP_PRIVATE, fd, 0);
 }

 void overflow(char *arg) {
        char buf[64];
        strcpy(buf, arg);
 }

 void move_stack(char *arg) {
        __asm("mov $0x04fffff00, %esp\n");
        overflow(arg);
        _exit(0);
 }

 int main(int argc, char *argv[]) {
        do_map_libc(); do_map_stack();
        move_stack(argv[1]);
 }
```

تصویر ۱۷: برنامه ی آسیب پذیر

نویسنده: **Hovav Shacham و دیگران**

مترجم (ترجمه ی آزاد): **سعید بیکی** (cephexin@secumania.net)

Secumania Security & Vulnerability Research Lab
www.secumania.net