جعل API و انواع روش‌های آن

سوسن نادری

soosan.naderi@gmail.com

آزمایشگاه تخصصی آپا در زمینه امنیت فن‌آوری اطلاعات و ارتباطات

http://cert.um.ac.ir

cert@um.ac.ir

ویرایش اول – آبان‌ماه ۱۳۹۲

شماره سند: APA_FUM_W_MAL_0106

چکیده

برای ایجاد تغییرات دلخواه در برنامه‌های کاربردی، غالبا نیاز به دسترسی به کد منبع آن است. با توجه به بسته بودن کد منبع اکثر برنامه‌های کاربردی، برنامه‌نویسان اغلب از روش‌های جعل API و یا تزریق DLL استفاده می‌کنند. در این مقاله چگونگی انجام آن‌ها مورد بررسی قرار می‌گیرد.

واژه‌های کلیدی

جعل API، تزریق DLL، وصله کردن، جدول آدرس‌های ورودی، جدول آدرس‌های خروجی، DLL پروکسی شده.

۱- مقدمه

برای استفاده از خدمات جدید برنامه‌هایی که توسط شرکت‌های تولیدکننده‌ی نرم‌افزار ارائه می‌شود، اغلب نیاز به در اختیار داشتن کد منبع برنامه‌ی اصلی است. برخی از برنامه‌های کاربردی برای توسعه، ابزارهای ارتباطی از قبیل افزونه‌ها و رابط‌های مشابه را فراهم می‌کنند. از آن‌جایی که کد منبع اکثر برنامه‌های کاربردی، به خصوص آن‌هایی که برای سیستم‌های عامل تجاری مانند ویندوز طراحی شده‌اند، به طور عمومی در دسترس نیست و یا مستندات آن‌ها ارایه نشده است، برنامه‌نویس امکان گسترش برنامه با تغییر کد منبع آن را ندارد. با این وجود، روش‌هایی وجود دارد که امکان گسترش برنامه را برای توسعه‌دهندگان فراهم می‌آورد. یکی از راه‌کارهای موجود این است که برنامه‌نویس می‌تواند از روش‌های برنامه‌نویسی سطح پایین مانند تغییر کد ماشین برنامه برای رسیدن به این هدف استفاده کند. در عمل، این روش برای رفع اشکالات کوچک، انجام تغییرات ناچیز و حتی دور زدن اقدامات امنیتی مورد استفاده قرار می‌گیرد. با این حال، هنگامی که اصلاحات بزرگتری برای یک برنامه مورد نیاز است، استفاده از برنامه‌نویسی در سطح کد ماشین به طور فزاینده‌ای دشوار می‌شود. با تمرکز بر روی سیستم عامل ویندوز می‌توان برای انجام مؤثرتر این کار از دو روش استفاده کرد:

۱. *تزریق DLL*[1] که به مفهوم بارگذاری کد در فضای آدرس برنامه‌ی مورد نظر از طریق یک تابع کتابخانه‌ای است. این امر فعل و انفعالات بعدی با حافظه‌ی برنامه و توابع را آسان‌تر می‌کند.

۲. *جعل API*[2] که روشی است برای تغییر رفتار برخی از توابع برنامه به گونه‌ای که از تابع دلخواه به جای تابع در نظر گرفته شده‌ی اصلی استفاده شود. از عبارت جعل کردن با عنوان جعل توابع نیز یاد می‌شود.

۲- تزریق DLL

تزریق DLL به معنی بارگذاری یک DLL دلخواه درون یک برنامه‌ی در حال اجرا است. این کار با اجرای کدی در برنامه از طریق API انجام می‌شود. این رویکرد، برای ساختار داده‌های مورد نیاز و تابع ایجادکننده‌ی نخ در پردازه هدف

[1] Dynamic Linked Library injection (DLL injection)
[2] Application Programming Interface hooking (API hooking)

حافظه اختصاص می‌دهد. برای انجام این کار، ابتدا با استفاده از تابع CreateRemoteThread یک نخ ایجاد شده و سپس، DLL دلخواه با استفاده از یک تابع API درون پردازه بارگذاری می‌شود.

تزریق DLL برای مقاصد مختلفی از قبیل ایجاد تغییرات در Windows Task Manager، جاسوسی در برنامه‌های کاربردی، انجام مجدد حملات تزریق کد و یا جلوگیری از آن استفاده می‌شود. همچنین، این روش برای ایزوله کردن برنامه‌های کاربردی در برخی از آنتی‌ویروس‌ها به کار گرفته می‌شود [۱].

۲-۱- روش‌های تزریق DLL

۲-۱-۱- ایجاد یک DLL پروکسی شده

DLL پروکسی شده DLLای است که جایگزین DLL اصلی می‌شود تا به وسیله‌ی آن فراخوانی‌های یک برنامه شنود شود، داده‌ها تغییر کند و حتی اشیای جدید ایجاد گردد [۲]. هنگامی که یک برنامه‌ی کاربردی نیاز به یک DLL دارد، ویندوز تلاش می‌کند که آن را در پوشه‌ها و یا مسیرهایی که اولویت‌بندی شده‌اند، جستجو کند [۳]. اگر DLL پروکسی شده را با اسمی مشابه DLL اصلی در یک پوشه با اولویت بالاتر قرار دهیم، DLL پروکسی شده به جای DLL اصلی بارگذاری می‌شود. از این پس، این DLL می‌تواند پیاده‌سازی‌های دیگری از توابع DLL اصلی را ارائه کند. اگرچه پیاده‌سازی این روش ساده است و به دانش برنامه‌نویسی کمی نیاز دارد، اما در مواردی که نیاز به تغییر مسیر تمام توابع درون DLL اصلی باشد، این روش مناسب نیست.

بسته به نوع پیاده‌سازی DLL اصلی، هرگونه تغییرات مانند به‌روزرسانی آن می‌تواند منجر به ایجاد مشکلاتی برای DLL پروکسی شده گردد. لازم به ذکر است که برخی نسخه‌های ویندوز مانند XP SP1 به بعد و ویندوز Vista در اولویت‌بندی جستجوی پوشه‌های DLL تغییراتی ایجاد کردند که باعث شده است پوشه‌ی برنامه‌ی کاربردی اجرا شده، اولویت کمتری نسبت به اولویت پوشه‌های سیستمی داشته باشد؛ در حالی که در نسخه‌های قدیمی‌تر ویندوز، پوشه‌ی برنامه‌ی در حال اجرا بالاترین اولویت را داشت. این اقدام تا حدودی اثربخشی و قابلیت استفاده از روش DLL پروکسی شده را در نسخه‌های اخیر ویندوز محدود کرده است [۱].

۲-۱-۲- اصلاح رجیستری ویندوز

در سیستم‌های عامل ویندوز مبتنی بر فن‌آوری NT، اکثر برنامه‌های کاربردی از user32.dll استفاده می‌کنند. استفاده از این تکنیک فرصتی را فراهم می‌آورد تا بتوان DLLهای مورد نیاز را به سیستم تزریق کرد. user32.dll در زمان اجرا، مقدار کلید AppInit_dll را خوانده و با استفاده از تابع Loadlibrary هنگامی که پردازه‌ی DLLProcessAttach را اجرا می‌کند، در پردازنده بارگذاری می‌کند [۴]. با توجه به این موضوع، می‌توان گفت که برای بارگذاری این DLLها نیاز به راه‌اندازی مجدد ویندوز می‌باشد که با توجه به آن، نقطه ضعف‌های بسیار زیادی برای استفاده از آن وجود دارد. اگرچه پیاده‌سازی این روش ساده است و به دانش برنامه‌نویسی کمی نیاز دارد، اما باعث می‌شود user32.dll در بیشتر برنامه‌های کاربردی که ممکن است مورد نیاز نباشد، بارگذاری شود [۵].

۲-۱-۳- جعل ویندوز

جعل ویندوز[1] یک مکانیزم تهیه شده توسط ویندوز است که در آن یک برنامه می‌تواند روی یک برنامه‌ی کاربردی خاص و یا *رویدادهای سیستم-گسترده*[2] (مانند رویدادهای ماوس یا صفحه‌کلید) نظارت داشته باشد. اگر جعل ویندوز سیستم-گسترده در یک DLL خاص ثبت شود، ویندوز (برای اکثر جعل‌ها) آن DLL را درون تمام برنامه‌های کاربردی که جعل کردن را راه‌اندازی می‌کنند، بارگذاری خواهد کرد (مثلاً رابط کاربری برنامه‌ی کاربردی، حرکات ماوس یا رویداد صفحه‌کلید را دریافت می‌کند).

اگرچه پیاده‌سازی این روش نیز ساده است و به دانش برنامه‌نویسی کمی نیاز دارد، DLL مورد نظر در تمام برنامه‌های کاربردی که جعل را راه‌اندازی می‌کنند، بارگذاری می‌شود. علاوه بر این، DLL تنها زمانی بارگذاری می‌شود که برنامه‌ی کاربردی جعل را راه‌اندازی کند. برای مثال، جعل WH_GETMESSAGE زمانی راه‌اندازی می‌شود که یکی از توابع GetMessage یا PeekMessage فراخوانی شوند. این موضوع اغلب برای برنامه‌های کاربردی با رابط کاربری گرافیکی پس از مقداردهی اولیه و وارد کردن *حلقه‌ی پیام*[3] اتفاق می‌افتد [۱].

[1] Windows hooks
[2] System-wide event
[3] Message loop

۲-۱-٤- استفاده از اشکال‌زدا

با استفاده از اشکال‌زدا[1] برای API، می‌توان نخی از یک برنامه‌ی کاربردی را برای اجرای یک کد دلخواه مانند بارگذاری یک DLL انتخاب کرد. این کار با ایجاد وقفه و بازنویسی کدی که هم‌اکنون در حال اجرا است، انجام می‌شود. بعد از این که اجرای کد جایگزین تمام شد، حالت اصلی برنامه بازیابی شده و اجرای آن از سر گرفته می‌شود. این روش قابل اعتماد است و کنترل بیشتری در فرآیند تزریق را فراهم می‌کند. با این حال، در صورتی که برنامه قبلاً اشکال‌زدایی شده باشد، این روش مناسب نیست [۱].

۲-۱-۵- وصله کردن جدول آدرس ورودی

وقتی مجوز برای نوشتن در فایل اجرایی وجود داشته باشد، این امکان وجود دارد که *جدول آدرس ورودی*[2] به خود فایل وصله شود تا DLL مورد نیاز برای تزریق در آن قرار گیرد. این کار نیاز به دانش اندکی در مورد *فرمت اجرایی قابل حمل*[3]، *تعمیر اجرایی قابل حمل*[4] و *جمع مقابله‌ای*[5] آن دارد. این روش اجازه‌ی تزریق اولیه را می‌دهد. اما مسأله‌ای که وجود دارد، اجبار به تغییر فایل اجرایی بر روی دیسک است. البته این کار امکان‌پذیر نیست و می‌تواند مقدار مقابله‌ای برنامه‌ی کاربردی مورد نظر را از بین ببرد. در مقابل، وصله کردن برنامه‌های کاربردی در حافظه به جای دیسک سخت معمولاً مقدار مقابله‌ای را دور می‌زند [۱].

۲-۱-٦- تزریق انعکاسی

از آن‌جا که تزریق DLL می‌تواند در تأمین امنیت مشکل ایجاد کند، برخی برنامه‌های کاربردی برای دفاع از خود در برابر آن از مکانیزم‌های خاصی بهره می‌برند. ویندوز فهرستی از تمام ماژول‌های بارگذاری شده را برای هر پردازه از قبیل ماژول برنامه‌ی اصلی و ماژول‌های DLL نگهداری می‌کند. اگر یک DLL از پردازه دیگری تزریق شود، ویندوز آن را در فهرستی ثبت می‌کند. این کار امکان تشخیص دست‌کاری و رفتار مطابق آن را برای برنامه‌ی کاربردی فراهم می‌کند.

[1] Debugger
[2] Input Address Table (IAT)
[3] PE format
[4] Fix-ups PE
[5] Checksum

تشخیص DLL تزریق شده با استفاده از این روش دشوار است. برای مثال، DLL تزریق شده نمی‌تواند با استفاده از EnumProcessModules (از کتابخانه‌ی PSAPI) کشف شود. تزریق انعکاسی DLL یک روش سطح پایین برای تزریق DLL می‌باشد که شامل اجرای حداقل یک *بارکننده‌ی اجرایی قابل حمل*[1] است. در این روش، DLL به گونه‌ای مخفی تزریق می‌شود و از دید برنامه‌ی مورد نظر مخفی باقی می‌ماند. روش‌های دیگری برای تشخیص تزریق وجود دارد که از طریق کنترل کردن اشاره‌گر دستورالعمل و نظارت بر محتوای حافظه انجام می‌شوند [۱].

۳- روش‌های جعل API

برنامه‌نویسان به جای ارتباط مستقیم با سیستم عامل، از APIهای ویندوز برای دسترسی به منابع سیستمی از قبیل فایل‌ها، پردازه‌ها، اطلاعات شبکه، رجیستری و غیره استفاده می‌کنند. حال اگر بتوان بر فراخوانی APIهای مربوطه و پارامترهای آن نظارت داشت، می‌توان آن را به صورت پویا تحلیل کرد. سیستم دایرکتوری ویندوز شامل APIهایی است که متشکل از چندین کتابخانه‌ی مهم از جمله kernel32.dll، ntdll.dll، ws2_32.dll و user32.dll می‌باشد.

برای مشاهده‌ی کنترل روند نمونه‌ای از بدافزار، باید بتوان به توابع موجود در APIها دسترسی داشت. یکی از راه‌های ممکن برای رسیدن به این هدف، جعل کردن توابع است.

کلمه‌ی جعل در برنامه‌نویسی کامپیوتر، طیف وسیعی از سازوکارهای مورد استفاده برای تغییر یا تکمیل رفتار یک سیستم عامل، یک برنامه‌ی کاربردی و یا سایر اجزای نرم‌افزارها را پوشش می‌دهد. این امر از طریق جلوگیری فراخوانی توابع یا پیام‌ها و یا رویدادهای موجود بین اجزای نرم‌افزار اتفاق می‌افتد. به کدی که مدیریت این وظایف را به عهده دارد، جعل گفته می‌شود.

دو نوع جعل وجود دارد: محلی و سراسری. جعل محلی تنها به یک برنامه‌ی خاص و جعل سراسری به تمام پردازه‌ها در سیستم اعمال می‌شود. جعل API به معنی متوقف کردن فراخوانی برخی از توابع موجود در رابط‌ها است که به وسیله‌ی آن می‌توان رفتار هر برنامه‌ای را تغییر داد. جعل به صورت گسترده توسط آنتی‌ویروس‌ها، برنامه‌های امنیتی، کاربردهای سیستمی، ابزارهای برنامه‌نویسی و موارد بسیار دیگر مورد استفاده قرار می‌گیرد.

[1] PE loader

هنگامی که یک برنامه تابعی را فراخوانی می‌کند، آن را به مکان دیگری که کد مورد نظر، جعل یا تابع جعل شده قرار دارد، مسیردهی مجدد می‌کند. سپس، جعل عملیات خود را انجام می‌دهد و کنترل را به تابع API برگردانده و یا به طور کامل مانع از اجرای آن می‌شود. اگر جعل به درستی انجام شود، تشخیص تابع جعل شده برای برنامه‌ی فراخوانی شده و همچنین، تشخیص این امر که تابع جعل شده به جای تابع اصلی فراخوانی شده است، سخت خواهد بود. در بحث مورد نظر ما که هدف آن استفاده از جعل برای تحلیل رفتار بدافزار است، باید توجه داشت که بدافزار می‌تواند سعی در تشخیص تابع جعل شده کند. بنابراین، جعل باید به دقت اجرا شود و سعی شود تا محیط تحلیل را تا آن‌جایی که امکان پذیر است از پردازه بدافزار پنهان کرد.

۳-۱- دستور پرش [٦]

ساده‌ترین روش جعل قرار دادن دستور پرش[1] است. مجموعه دستورات معماری x86 دارای اندازه متغیر بین یک بایت و حداکثر شانزده بایت هستند. برای مثال، یک دستور خاص مانند پرش غیرشرطی، پنج بایت طول دارد. یک بایت نشان‌دهنده‌ی *کد عملگر*[2] و چهار بایت دیگر نشان‌دهنده‌ی آفست نسبی ۳۲ بیتی است (البته باید توجه داشت که یک دستور پرش غیرشرطی نیز وجود دارد که آفست نسبی ۸ بیتی دارد).

اگر دو تابع A و B وجود داشته باشد، می‌توان با استفاده از دستور پرش، روند اجرای برنامه را از تابع A به تابع B تغییر مسیر داد. برای این کار تنها کافی است آفست نسبی درست را محاسبه کرد.

فرض می‌کنیم تابع A در آدرس 0x401000 و تابع B در آدرس 0x101800 واقع شده باشد. آن‌چه باید انجام داد، تعیین آفست نسبی مورد نیاز است. اگر در ابتدای تابع A که در آدرس 0x401000 قرار دارد، دستور پرش گذاشته و اجرا شود، کاری که CPU انجام خواهد داد به این شرح است: ابتدا طول دستور پرش که پنج بایت است را به *اشاره‌گر دستورالعمل*[3] اضافه می‌کند. سپس، آفست نسبی به اشاره‌گر دستور اضافه می‌شود (چهار بایت یا مقدار ۳۲ بیتی، بعد از کد عملگر قرار دارد). به بیان دیگر، CPU اشاره‌گر جدید دستور را مشابه زیر محاسبه می‌کند:

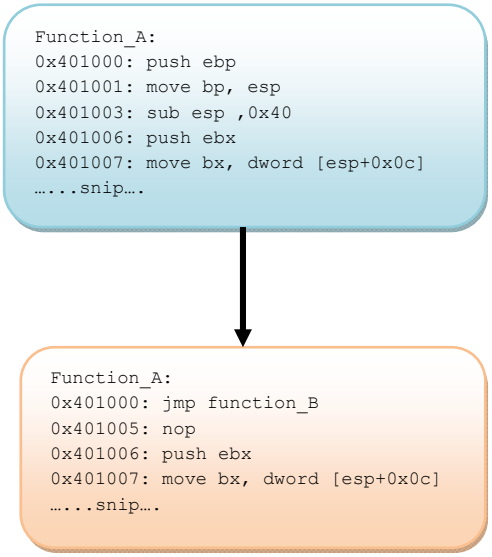
```
Instruction_pointer = instruction_pointer + 5 + relative_offset ;
```

بنابراین، محاسبه‌ی آفست نسبی ما را ملزم به معکوس کردن فرمول به روش زیر می‌کند:

[1] Jump
[2] Opcode
[3] Instruction pointer

Relative_offset = function_B – function_A – 5 ;

پنج بایت کم شده به دلیل مقدار طول دستور پرش است که CPU هنگام اجرای این دستور اضافه می‌کند. علت کم شدن function_A از function_B، نسبی بودن پرش است. تفاوت بین آدرس‌های function_B و function_A، 0x800 بایت است. اگر فراموش شود آدرس function_A از آدرس function_B کم شود، آن‌گاه CPU در آدرس 0x401800 + 0x401000 + 5 متوقف خواهد شد، که بدیهی است که این آدرس مورد نظر ما نیست. در زبان اسمبلی، تغییر مسیر تابع A به تابع B تقریباً مشابه شکل (۱) است:



شکل (۱) : تغییر مسیر تابع A به تابع B

قبل از جعل، چند دستور اصلی دیده می‌شود؛ هرچند که بعد از عمل جعل کردن، آن دستورات توسط دستور پرش بازنویسی می‌شوند. همان‌طور که در شکل (۱) دیده می‌شود، سه دستور اول در مجموع شش بایت طول دارند (برای نمونه می‌توان دید که دستور push ebx در آدرس 0x401006 قرار دارد). در این‌جا دستور پرش تنها پنج بایت استفاده می‌کند و یک بایت باقی‌مانده با دستور nop جایگزین می‌شود.

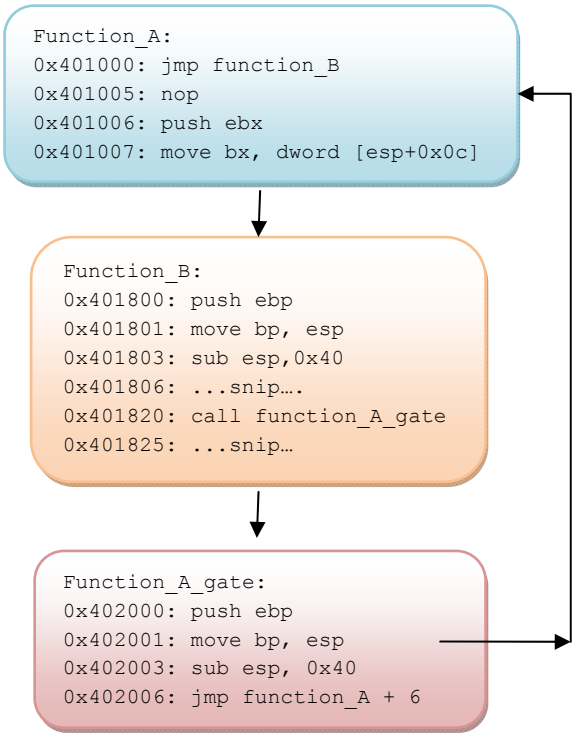
۳-۱-۱- Trampoline

می‌توان مسیر تابع A که جعل شده است را به تابع B تغییر داد. برای مشهود بودن این که چگونه تابع اصلی A بدون اجرای عملیات جعل اجرا می‌شود، باید تابعی به نام Trampoline ایجاد کرد. قطعه کد زیر مثال ساده‌ای است که روش

استفاده‌ی Trampoline برای تابع جعل شده را نشان می‌دهد. در این مثال، function_A_Trampoline نشان‌دهنده‌ی Trampoline برای تابع جعل شده‌ی A است.

```
voidfunction_A ( int value , int value2) ;
    void (*function_A_Trampoline)(int value, int value2);
    voidfunction_B(intvalue, intvalue2){
    function_A_Trampoline(value + 1, value2 + 2); }
```

در مثال فوق، پنج بایت اول تابع A مجدداً نوشته می‌شود. برای اجرای تابع اصلی بدون جعل باید بایت‌هایی که دوباره‌نویسی شده‌اند را هنگام نصب جعل اجرا کرد؛ سپس، به آدرس تابع A که چند بایت به آن اضافه شده است، پرش کرد. این دقیقاً همان چیزی است که در قطعه کد بالا اتفاق می‌افتد. شکل (۲) عملکرد داخلی Trampoline را نشان می‌دهد:



شکل (۲) : عملکرد داخلی Trampoline

در شکل (۲) روند اجرا به این صورت است: تابع A فراخوانی شده، جعل اجرا می‌شود و سپس، ادامه‌ی اجرا از تابع B است. تابع B دستوراتی را اجرا می‌کند، اما در آدرس 0x401820 می‌خواهد تابع اصلی الف را بدون جعل اجرا کند؛ این‌جا است که Trampoline مطرح می‌شود. Trampoline شامل دو قسمت است: دستورات اصلی و پرش به تابع

A. در واقع با توجه به تعاریف بالا می‌توان گفت که Trampoline قطعه کدی است که به دلیل استفاده از دستور پرش حذف شده است و باید برای اجرای صحیح در روند برنامه بازنویسی شود. پرش در Trampoline با استفاده از فرمولی که قبلاً گفته شد، محاسبه می‌شود؛ اگرچه در این مورد خاص آدرس‌ها و آفست‌ها کمی متفاوتند. بنابراین، فرمول دقیق به صورت زیر است:

```
relative_offset= (function_A_trampoline + 6) - (function_A + 6) - 5;
```

باید به دقت توجه داشت که در این‌جا از (function_A_trampoline + 6) 0x402006 به (function_A + 6) 0x401006 پرش شده است.

۳-۱-۲- تشخیص استفاده از توابع جعل شده

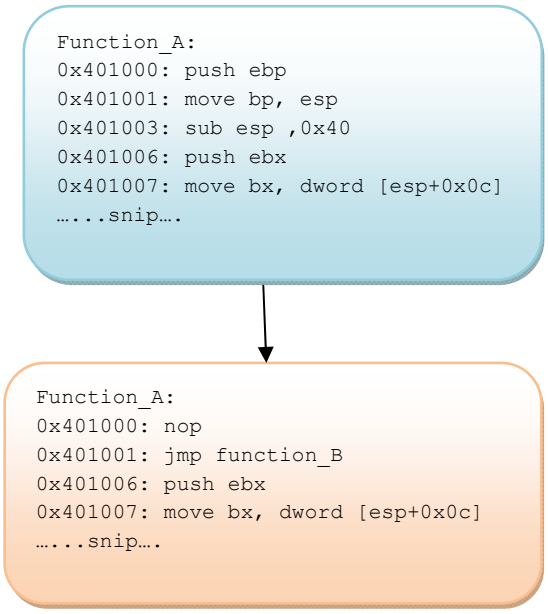
از آن‌جا که این روش جعل کردن بسیار ساده است (قرار دادن یک دستور پرش)، تشخیص آن نیز چندان مشکل نیست. تشخیص جعل در این مثال به این صورت است:

```
if(*function_A == 0xe9) {
printf("Hook detected in function A.\n");}
```

0xe9 کد عملگر برای پرش به همراه آفست نسبی ۳۲ بیتی است.

۳-۲- Prepend a Nop [٦]

به جای نوشتن دستور پرش در تابع A، می‌توان ابتدا یک دستور nop را نوشت و به دنبال آن دستور پرش را آورد. هنگام استفاده از این روش، باید توجه داشت که در حال حاضر دستور پرش در آدرس (function_A + 1) 0x401001 قرار دارد که آفست نسبی را به اندازه‌ی یک واحد تغییر می‌دهد.



شکل (۳) : روش prepend a nop

از آن‌جایی که اولین دستور در تابع A دستور nop است، روش تشخیص به صورت زیر خواهد بود:

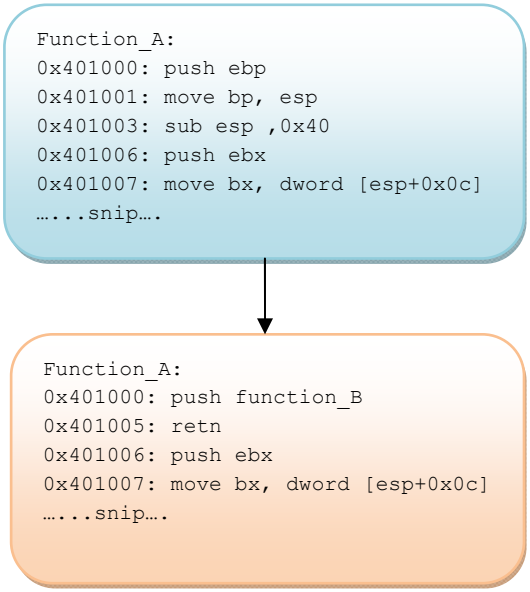
```
unsigned char*addr = function_A;
while(*addr == 0x90) addr++;
if(*addr == 0xe9) {
printf("Hook detected in function A.\n");
}
```

همان‌طور که مشاهده می‌شود، بعد از عبور از تمام دستورات nop که کد عملگر آن‌ها 0x90 است، دستور پرش مورد بررسی قرار می‌گیرد.

[۶] Push / Retn -۳-۳

دستور push یک مقدار ۳۲ بیتی روی پشته قرار می‌دهد. سپس، دستور retn یک آدرس ۳۲ بیتی را از روی پشته برمی‌دارد و در اشاره‌گر دستور قرار می‌دهد. به بیان دیگر، اجرا از آدرسی که در بالای پشته قرار دارد، شروع می‌شود.

این روش در مجموع شش بایت است. توجه به این نکته ضروری است که دستور push یک آدرس مستقیم می‌گیرد، نه یک آدرس نسبی.



شکل (٤) : روش push / retn

تشخیص این روش به صورت زیر است. اگرچه باید به خاطر داشت که دستور nop پیشوندی یا گذاشتن یک nop بین دستورات push و retn از طریق کد زیر تشخیص داده نمی‌شود:

```
unsigned char*addr = function_A;
if(*addr == 0x68 &&addr[5] == 0xc3) {
printf("Hook detected in function A.\n");
}
```

0x68 کد عملگر دستور push و 0xc3 کد عملگر دستور retn است.

۳-٤- نقاط شناور [٦]

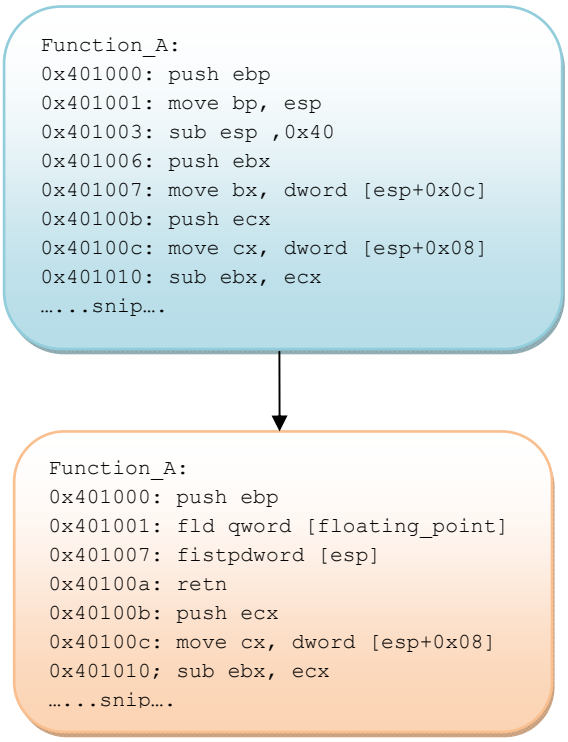
روش‌هایی که تاکنون گفته شد روی دستورات معماری x86 متمرکز بودند. روش جالب دیگری نیز وجود دارد که مربوط به *نقاط شناور*[1] است.

این روش مانند روش push / retn است که در آن یک مقدار ساختگی روی پشته قرار می‌گیرد و سپس این مقدار ساختگی با آدرس واقعی بازنویسی شده و سپس برگردانده می‌شود.

[1] Floating points

نکته‌ی جالب در این روش این است که به جای ذخیره‌سازی آدرس پرش به صورت یک آدرس ۳۲ بیتی، این آدرس به صورت یک نقطه‌ی شناور ۶۴ بیتی ذخیره می‌شود. سپس، آدرس پرش با استفاده از دستور fld خوانده شده و با استفاده از دستور fistp به یک مقدار ۳۲ بیتی ترجمه می‌شود.

شکل (۵) این روش را نشان می‌دهد. باید در نظر داشت که جعل، ۱۱ بایت استفاده می‌کند. همچنین، floating_point یک اشاره‌گر به مقدار نقطه‌ی شناور ۶۴ بیتی است که شامل آدرس تابع جعل شده می‌باشد.



شکل (۵) : روش نقطه‌ی شناور

ساخت نقطه‌ی شناور نسبتاً آسان و به صورت زیر است:

doublefloating_point_value = (double) function_B;

در این‌جا تابع B، تابع جعل است.

ساخت نقطه‌ی شناور که همان فراخوانی تابع اصلی از طریق یک Trampoline است، مانند روش‌های دیگر انجام می‌شود (به جز این مورد که در آن Trampoline شامل حداقل دستوراتی است که در ۱۱ بایت اول تابع قرار دارد).

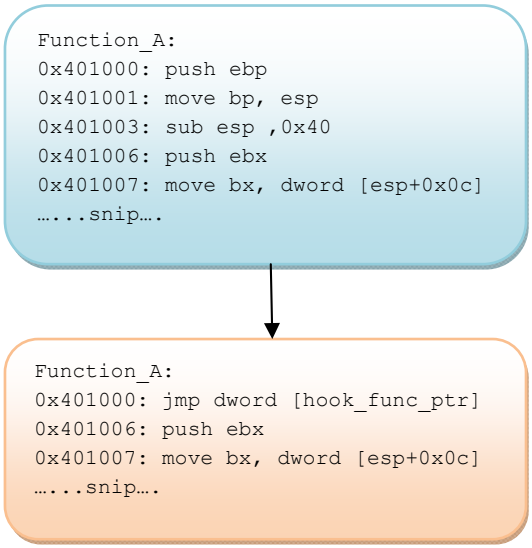
۳-۵- MMX / SSE [٦]

این تکنیک مشابه جعل کردن با استفاده از نقطه‌ی شناور است. اما در این روش به جای استفاده از نقطه‌ی شناور، از پسوندهای MMX یا SSE استفاده می‌شود.

هر دو روش مانند روش نقطه‌ی شناور از متد push / retn استفاده می‌کنند. روش اول شامل دستورات MMX است که اغلب از دستور movd استفاده می‌کند. این کار درست مثل دستور fistp (دستور نقطه‌ی شناور) اجازه می‌دهد یک مقدار را از حافظه خوانده و یک مقدار را در پشته ذخیره کند. روش دوم، استفاده از دستورات SSE است که از همان دستور movd استفاده می‌کند. تنها تفاوت بین این دو روش این است که دستورات MMX در ثبات‌های ۶۴ بیتی و دستورات SSE در ثبات‌های ۱۲۸ بیتی کار می‌کنند (باید توجه داشت که دستور movd تنها اجازه‌ی خواندن و نوشتن مقادیر ۳۲ بیتی را دارد). در هر صورت، این روش‌ها دقیقاً مانند روش نقطه‌ی شناور هستند، تنها تفاوتشان در دستوراتی است که استفاده می‌کنند.

۳-٦- پرش غیرمستقیم [٦]

پرش غیرمستقیم می‌گوید به آدرسی که در این‌جا است پرش کن. یک پرش غیرمستقیم، شش بایت طول دارد.



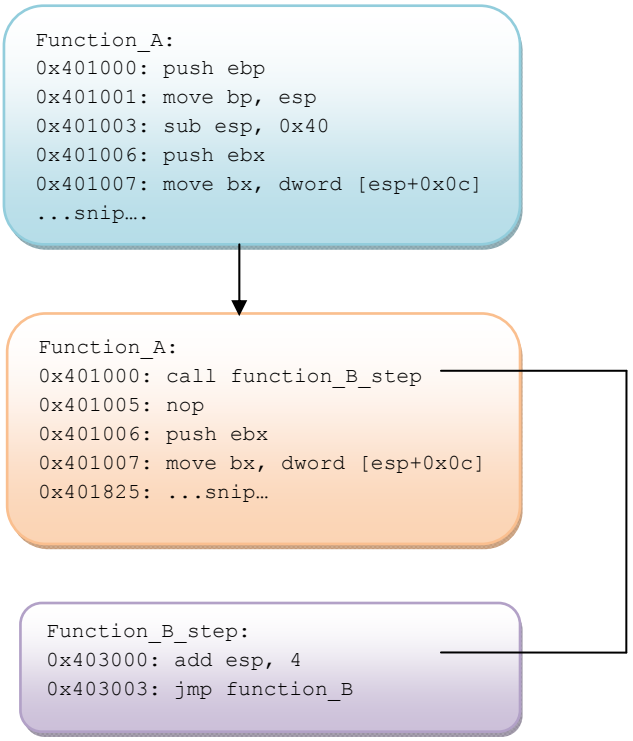
شکل (٦) : روش پرش غیرمستقیم

توجه داشته باشید که hook_func_ptr نشان‌دهنده‌ی آدرس تابع جعل (برای نمونه تابع B) است.

۳-۷- دستور فراخوانی [٦]

در حالی که همه‌ی روش‌های جعل گفته شده مستقیماً به خود تابع جعل (برای نمونه تابع B) پرش می‌کنند، این روش نیاز به یک گام اضافی دارد. دلیل این کار این است که دستور call پس از قرار دادن آدرس بازگشت در پشته، به یک آدرس مشخص پرش می‌کند. آدرس بازگشت، اشاره‌گر دستور فعلی به علاوه‌ی طول دستور فراخوانی است.

همان‌طور که در مثال زیر مشاهده می‌شود، یک آدرس بازگشت اضافی بر روی پشته وجود دارد. اول باید این آدرس را از پشته برداشت؛ در غیر این صورت، پشته خراب خواهد شد (برای نمونه، با توجه به این که اشاره‌گر پشته نادرست است، تابع اتصال پارامترهای نادرست را از پشته می‌خواند). وقتی که آدرس در پشته گذاشته می‌شود، ابتدا اشاره‌گر پشته عدد چهار را از آن کم می‌کند. سپس، آدرس در آدرسی که اشاره‌گر به‌روزرسانی شده‌ی پشته به آن اشاره می‌کند، نوشته می‌شود. با اضافه کردن عدد چهار به اشاره‌گر پشته می‌توان این آدرس را از پشته برداشت. پس از این که آدرس از پشته برداشته شد، به تابع جعل شده پرش می‌شود. این سازوکار برای هر دو نوع دستور call مستقیم و غیرمستقیم عمل می‌کند. تصویر زیر نشان‌دهنده‌ی این موضوع است:



شکل (۷) : روش دستور فراخوانی

۳-۸- جعل کردن از طریق وصله کردن جدول آدرس ورودی [۱،۷]

پایه و اساس روش وصله کردن جدول آدرس ورودی، متکی بر این واقعیت است که فایل‌های اجرایی ۳۲ بیتی مبتنی بر سیستم عامل ویندوز و توابع کتابخانه‌ای، بر اساس فرمت فایل اجرایی قابل حمل ساخته شده‌اند. فایل‌هایی که این خصوصیات را دارند، از بخش‌های مختلفی تشکیل شده‌اند که هر بخش شامل اطلاعات خاصی است. برای نمونه، بخش text. دارای کد کامپایل شده از برنامه است، در حالی که بخش rsrc. به عنوان یک مخزن برای منابعی از قبیل دیالوگ‌ها[1]، تصاویر Bitmap و *نوار ابزار*[2] است.

در میان تمام بخش‌های یک فایل اجرایی ویندوز، بخش idata. برای پیاده‌سازی *جداساز API*[3] بسیار مفید است. در این بخش، یک جدول خاص به نام جدول آدرس‌های ورودی قرار دارد که دارای *آفست‌های نسبی فایل*[4] برای نام توابع ورودی است که از طریق کد اجرایی به آن‌ها ارجاع داده می‌شود. هرگاه ویندوز یک فایل اجرایی را در حافظه بارگذاری می‌کند، این آفست‌ها را با آدرس‌های تصحیح شده نسبت به آدرس پایه‌ی توابع ورودی وصله (بازنویسی) می‌کند.

در پیاده‌سازی فعلی فایل‌های اجرایی ویندوز و توابع کتابخانه‌ای، مسیر فراخوانی‌هایی که به توابع ورودی اعمال می‌شود، توسط جدول آدرس ورودی با استفاده از دستور پرش غیرمستقیم تعیین می‌شود. هنگامی که فایل اجرایی نیاز به فراخوانی تابع دارد، به سادگی به دنبال آدرس تابع در جدول آدرس ورودی خود می‌گردد. از طریق وصله کردن جدول آدرس ورودی، امکان جعل کردن فراخوانی یک تابع به تابع جعل شده وجود دارد.

۳-۹- وصله کردن جدول آدرس خروجی [۱،۷]

وصله کردن *جدول آدرس خروجی*[5] مشابه با وصله کردن جدول آدرس ورودی است، با این تفاوت که به جای جدول آدرس ورودی فایل اجرایی، جدول آدرس خروجی توابع کتابخانه‌ای وصله می‌شود. جدول آدرس خروجی توابع کتابخانه‌ای حاوی آدرس تمام توابعی است که DLL استفاده می‌کند. وصله کردن جدول آدرس ورودی فایل اجرایی یا

[1] Dialog boxes
[2] Toolbars
[3] API interceptor
[4] File-relative offset
[5] Export Address Table (EAT)

جدول آدرس خروجی توابع کتابخانه‌ای با پیوند پویا نتیجه‌ی یکسانی تولید می‌کند. تفاوت اصلی که باعث شده این روش جعل کردن در مقایسه با روش جعل کردن جدول آدرس ورودی بیشتر مورد استفاده قرار گیرد، این است که بر روی تمام برنامه‌هایی که از DLL استفاده می‌کنند، تأثیر می‌گذارد.

وصله کردن جدول آدرس خروجی کار دشواری نیست و نتایج جالبی دارد. در این روش تمام پیوندهای جدید اعم از ایستا و پویا که مربوط به API وصله شده‌اند، به طور خودکار به جای تابع اصلی به تابع فراخوانی شده مرتبط می‌شوند.

۳-۱۰- جعل کردن از طریق پروکسی توابع کتابخانه‌ای [۵،۷]

امکان جعل API به هنگام تزریق DLL و با استفاده از روش DLL پروکسی شده نیز وجود دارد. در این روش، DLL مورد نظر با DLLای با همان نام جایگزین شده و تمام توابع DLL اصلی خارج می‌شوند. DLL پروکسی شده حاوی یک *نقطه‌ی ورودی*[1] به نام *تابع فرستنده*[2] است [۱] که فراخوانی یک تابع را به تابع دیگری در آن DLL محول می‌کند. علاوه بر امکان فراخوانی DLL اصلی برای انجام یک سری عملیات، ممکن است پیاده‌سازی‌های جایگزین از این توابع ارائه شود.

٤- نتیجه‌گیری

در این مقاله، چگونگی انجام تزریق توابع کتابخانه‌ای و همچنین انواع روش‌های آن‌ها بیان شد. هنگامی که برنامه‌نویسان نیاز به اصلاح برنامه‌های شخص ثالث دارند، غالباً با استفاده از جعل API قادر به اعمال تغییرات دلخواه بدون ایجاد تغییر و یا حتی داشتن دسترسی به کد منبع برنامه می‌باشند. لذا، در این مقاله چگونگی انجام جعل API و همچنین انواع روش‌های آن‌ها مورد بررسی قرار گرفت.

[1] Entry point
[2] Function forwarder

مراجع

[1] Extending applications using an advanced approach to DLL injection and API hooking . Bosnic, J. Berdajs and Z 2010. Wiley InterScience.

[2] Create your Proxy DLLs automatically .Codeproject 2013 http://www.codeproject.com/Articles/16541/Create-your-Proxy-DLLs-automatically.

[3] MSDN .Dynamic-Link Library Search Order .*MSDN.* Microsoft, 26 10 2012 http://msdn.microsoft.com/en-us/library/windows/desktop/ms682586(v=vs.85).aspx.

[4] Working with the AppInit_DLLs registry value .*MSDN* .Microsoft, 21 11 2006 http://support.microsoft.com/kb/197571.

[5] API hooking revealed .*CodeProject* www.codeproject.com\API hooking revealed.

[6] *Toward Automated Dynamic Malware Analysis Using CWSandbox* .CARSTEN WILLEMS, THORSTEN HOLZ,FELIX FREILING 2007, IEEE SECURITY & PRIVACY.

[7] Breme, Jurriaan .*www.google.com.* www.google.com\x86 API Hooking Demystified _ Development & Security.htm.

[8] *www.*madshi.*net.* www.madshi.net\ApiHookingMethods.htm.

[9] *www.amazon.com.* www.amazon.com\apispy.htm.

[10] Richter, Jeffrey .*Programming Application for MS Windows* .Redmond, Washington : Microsoft Press, 1999.