معرفی Memory Injection و ایده‌هایی برای جلوگیری از این گونه حملات

فهرست مطالب

معرفی

روزانه بر روش‌های تزریق کد به فرایندهای در حال اجرا افزوده می‌شود و معروف ترین و مؤثرترین روشی که تا به حال مورد استفاده قرار گرفته است، Reflective Memory Injection نام دارد.

هر کدام از روش های به کار رفته دارای ویژگی‌ها و تکنیک های خاص خود می‌باشند و تنها نقطه مشترکی که می‌توان به آن اشاره کرد که تقریبا در همه روش‌ها یکسان مورد استفاده قرار گرفته است، API هایی است که مهاجمان برای تزریق و اجرای کد خود از آن‌ها استفاده می‌کنند. و نکته جالب تر این است که تمام این API ها مربوط به خود OS قربانی است و این API ها به نوعی مجاز برای استفاده هستند در دسترس تمام برنامه‌ها.

تزریق dll در فرایندهای در حال اجرا

برخی توابع تعریف شده در ویندوز API به صورت مستقیم به مهاجمان اجازه می‌دهند که عمل تزریق dll در دیگر فرایندها را انجام دهند. یعنی می‌توان گفت که تزریق dll یک عمل پزیرفته شده توسط سازندگان ویندوز است و این روش در خیلی مواقع به صورت کاملا مجاز مورد استفاده قرار می‌گیرد. اما این توابع آسیب پذیری جدی در سیستم به وجود آورده‌اند که به مهاجمان نیز این اجازه را می‌دهند تا از این APIها برای اهداف مخربانه خود استفاده کنند.

تزریق dll روش های مختلفی دارد که که همچنان نیز در حال توسعه ابداع روش های جدید می‌باشد. اما می‌توان گفت که تقریبا همه‌ی این روش‌ها از یک مسیر مشخص باید عبور کنند که در ادامه ذکر شده است.

1. چسبیدن به فرایند قربانی
2. اختصاص دادن حجمی از حافظه در فرایند مهاجم به مقداری مشخص و مورد نیاز
3. کپی کردن dll در فرایند قربانی و مشخص کردن آدرس‌های توبع مورد نیاز از dll بارگذاری شده در حافظه
4. اجرای dll با آموختن نحوه اجرای dll به فرایند قربانی

در تکنیک‌های مختلف استفاده شده در تزریق dll ممکن است که این مراحل به شیوه‌های مختلفی پیاده‌سازی شوند و یا حتی برخی مراحل اضافه و یا حذف شوند اما عمدتا این مراحل برای کامل کردن مراحل تزریق لازم هستند.

در ادامه API های رایجی که در هر مرحله مورد استفاده قرار می‌گیرند به صورت تفکیک شده معرفی می‌شوند و یک مثال از یک تکنیک خاص و مؤثر در ادامه مطالب قرار داده شده است.

مراحل تزریق و رایج ترین API های استفاده شده در این مراحل

چسبیدن به فرایند قربانی

در اکثر مواقع برای انجام مرحله‌ی اول در تزریق dll از تابع موجود در API به نام OpenProcess استفاده می‌شود که با دادن مقدار شناسه فرایندی مشخص(قربانی) یک handle از فرایند برمی‌گرداند.

در این مرحله ممکن است نتوان تمام فرایندهای در حال اجرا را به این روش در معرض تزریق قرار داد به این دلیل که فرایندهای مختلف با سطوح دسترسی مختلف در سیستم ثبت می‌شوند که برخی فقط از سوی خود سیستم مورد استفاده قرار می‌گیرند و دیگر برنامه‌ها قادر به استفاده از این روش در چسبیدن به این گونه فرایندها نیستند. البته با برخی تکنیک ها می‌توان تا مقداری سطح دسترسی فرایند مهاجم را بالا برد تا بتواند به فرایندهای با سطح دسترسی بالاتر دست پیدا کند. اما افزایش سطح دسترسی تا حدی مجاز است و باز نمی توان به همه‌ی فرایندها دسترسی پیدا خصوصا فرایندهایی که موبوط به سیستم عامل می‌باشند.

به منظور گرفتن اجازه دسترسی پیدا کردن به فرایندهای با سطوح دسترسی بالاتر، از توابع API و به شکل زیر اقدام می‌شود:

```
void EnableDebugPriv()
{
    HANDLE hToken;
    LUID luid;
    TOKEN_PRIVILEGES tkp;

    OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken);
    LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &luid);
    tkp.PrivilegeCount = 1;
    tkp.Privileges[0].Luid = luid;
    tkp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
    AdjustTokenPrivileges(hToken, false, &tkp, sizeof(tkp), NULL, NULL);

    CloseHandle(hToken);
}
```

با انجام این فرایند یک اجازه نامه برای فرایند هدف، که در اینجا فرایند فراخواننده این تابع است، صادر شده و از این پس فرایند می‌تواند به فرایندهای بیشتری دسترسی داشته باشد.

برای پیدا کردن فرایندهای در حال اجرا از توابع دیگری استفاده می‌شود که با استفاده از آن‌ها می‌توان به اطلاعات تمامی فرایندهای در حال اجرا دسترسی داشت. این توابع عبارت‌اند از:

- `CreateToolhelp32Snapshot`

  این تابع اصلی به کار رفته برای یافتن شناسه فرایندهای در حال اجرا می‌باشد.

  از این تابع می‌توان برای شناسایی فرایندها و زیر فرایندها، ماژول‌های مربوط به هرکدام از فرایندها، نخ های ایجاد شده برای هر کدام از فرایندها و ماژول‌ها و همچنین لیستی از اطلاعات مربوط به حافظه‌های هیپ را به دست آورد.

  این تابع حتی می‌تواند اطلاعاتی از جمله: آدرس پایه موارد ذکر شده، حجم آنها و صاحب آنها و اسم موارد ذکر شده را نشان دهد و می‌توان با این تابع dll های مورد استفاده قرار گرفته توسط هر فرایند را شناسایی کرد.

اختصاص حجم مورد نیاز از حافظه برای بارگذاری dll

در این مرحله باید حجم مورد نیاز برای بارگذاری کامل dll محاسبه شود و بعد dll را در محلی از حافظه که به حجم مورد نیاز اختصاص داده شده است منتقل شود و یک handle از dll تازه بارگذاری شده دریافت کنیم تا در مراحل بعدی بتوانیم دوباره از dll استفاده کنیم.

نکته : تابع LoadLibrary به صورت رایج برای انجام تزریق در بیشتر روش‌ها استفاده می‌شود که در حملات نوع جدید دیگر استفاده از این تابع به دلیل شناخته شدن بسیار کم‌رنگ شده است. این تابع می‌تواند یک dll را به صورت مستقیم در یک فرایند بارگذاری کند و خیلی از مراحل زیر را خلاصه کند.

برای محاسبه حجم مورد نیاز dll باید اول dll را باز کرد تا بتوان اطلاعات درون آن را خواند و به حجم مورد نیاز آن پی برد برای این کار معمولا از تابع CreateFile استفاده می‌شود که این تابع در صورت وجود فایل درخواستی یک handle از آن برمی‌گرداند و به ما اجازه می‌دهد تا اطلاعات داخل فایل را بخوانیم خب در این مرحله dll هدف را باز می‌کنیم و در مرحله‌ی بعد سایز مورد نیاز dll را توسط تابع GetFileSize به دست می‌آوریم که کار نسبتا راحتی است اما این کار را می‌توان با استفاده از خواندن اطلاعات داخل هیدر های PE مروبط به dll نیز به دست آورد که نسبتا دشوار تر از استفاده از این تابع است.

بعد از محاسبه‌ی حجم مورد نیاز dll به اصلی ترین مرحله یعنی بارگذاری dll می‌رسیم که به شیوه‌های مختلفی می‌توان آن را پیاده سازی کرد. به طوری که، این کار را می‌توان در هر قسمتی از حافظه که برای فرایند مهاجم قابل دسترس و دارای سطح دسترسی write است انجام داد. معمولا این کار در مکان‌هایی از حافظه انجام می‌شود که دارای سطح دسترسی RW می‌باشد و نه دسترسی RWX. به این منظور بارگذاری اولیه‌ی dll معمولا بر روی حافظه هیپ فرایند مهاجم انجام می‌شود و توسط تابع HeapAllocو دیگر توابعی که به منظور بارگذاری در حافظه انجام می‌شوند و این توابع معمولا با کلمه کلیدی Alloc شناخته می‌شوند. مخفف کلمه Allocation و به معنی اختصاص دادن می‌باشد. (مانند virtualAlloc، malloc، virtualAllocEx و ...)

نکته : بعد از بارگذاری dll بر روی حافظه باید بتوان با استفاده از handle به دست آمده بتوان dll را از حافظه خواند، در غیر این صورت dll به درستی بارگذاری نشده است.

کپی کردن dll در حافظه فرایند قربانی و مشخص کردن آدرس‌ها

تفاوت اصلی بین روش‌های تزریق کد در مرحله و مرحله بعدی نمایان می‌شود اما تقریبا همگی یک هدف را دنبال می‌کنند یعنی تزریق dll.

در یکی از روش‌های مؤثر به نام dll reflective injection (قدم های مربوط به این روش در ادامه به صورت مختصر تشریح خواهد شد)

در این روش در خود dll ی که قصد تزریق آن را داریم یک تابع وجود دارد که خودش کار تنظیمات خود آماده سازی خود را انجام می‌دهد که وارد جزئیات آن نمی‌شویم. کار مهاجم در این روش این است که اول آدرس نسبی این تابع را در dll پیدا کند و که این کار با خواندن اطلاعاتی از روی هیدر های PE فایل dll صورت می‌گیرد و بعد از پیدا کردن آدرس نسبی این تابع مراحل اصلی تزریق را آغاز می‌کنیم که با بارگذاری dll بر روی فرایند قربانی شروع می‌شود که توسط تابع virtualAlloc معمولا انجام می‌شود و بعد از بارگذاری فایل باید با استفاده از تابع writeProcessMemory فایل بارگذاری شده در حافظه‌ی را به حفظه‌ی اصلی مربوط به فرایند قربانی درج کرد.

اجرای dll با آموختن نحوه اجرای dll به فرایند قربانی

در آخرین مرحله با استفاده از تابع CreateRemoteThread برای اجرای تابع dll_main موجود در داخل dll تزریق شده، یک نخ جدید در فرایند قربانی ایجاد می کنیم تا عملکرد مورد نظر مهاجم در فرایند قربانی، به صورت یک نخ جدا، اجرا شود. در واقع در روش reflective آدرس یک تابع به نخ داده شده و این تابع که درون خود dll تعریف شده است خود dll را به فرایند قربانی متصل می‌کند و توابع export و import را که نیاز است به اصطلاح resolve می‌کند و که بتواند dll را آماده اجرا کند و بعد از اتمام این کار dll_main در dll اجرا می‌شود.

مراحل انجام reflective dll injection با تشریح مراحل

بارگزاری و خواندن dll

برای خواندن محتوای dll مورد نظر باید توسط تابع زیر یک کنترل از dll را ایجاد و ذخیره کنیم تا در مراحل بعدی بتوانیم اطلاعات مورد نیاز را از این کنترل بدست آورد

```
hFile = CreateFileA( cpDllFile, GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL );
```

cpDLLFile نام dll مورد نظر می‌باشد. GENERIC_READ مشخص می‌کند که فایل به صورت فقط قابل خواندن باشد. پارامتر سوم مشخص وضعیت اشتراک گذاری فایل را مشخص می‌کند که در اینجا ما از اشتراک گذاری فایل جلوگیری کرده‌ایم. پارامتر چهارم اشاره‌گری به ساختار SECUTITY_ATTRIBUTES است که با گذاشتن مقدار NULL ویژگی‌های امنیتی پیشفرض برای فایل مورد استفاده قرار می‌گیرند. با مشخص کردن مقدار OPEN_EXISTING مشخص می‌شود که فایل فقط در صورت وجود داشتن باز خواهد شد. با دادن مقدار FILE_ATTRIBUTE_NORMAL ما ویژگی‌های معمولی برای باز کردن فایل را استفاده کرده‌ایم. پارامتر آخر برای استفاده از فایل Template است که ما در اینجا از این گزینه استفاده نکرده‌ایم.

با استفاده از تابع بعدی باید حجم مورد نیاز فایل dll را بدست آوریم

```
dwLength = GetFileSize( hFile, NULL );
```

و پس از فهمیدن حجم مورد نیاز dll یک بخش از حافظه‌ی Heap فرایند فعلی در حال اجرا را برای بارگذاری dll تخصیص می‌دهیم.

```
lpBuffer = HeapAlloc( GetProcessHeap(), 0, dwLength );
```

در مرحله‌ی بعد فایل را بارگذاری کرده و اطلاعات آن را از روی حافظه می‌خوانیم و در متغیر dwBytesRead ذخیره می‌کنیم.

```
ReadFile( hFile, lpBuffer, dwLength, &dwBytesRead, NULL )
```

ایجاد کنترل از فرایند هدف

هر کدام از فرایند ها دارای سطوح دسترسی متفاوتی برای انواع کاربرهای سیستم عامل می‌باشند. برای اینکه بتوان از فرایند هدف کنترل ایجاد کرد باید اول سطح دسترسی کاربر را ارتقا دهیم که برای این کار به روش زیر عمل می‌شود و با گرفتن token قبلی که مشخص کننده‌ی سطوح دسترسی کاربر در بخش‌های مختلف سیستم عامل است این کار را انجام می‌هیم.

```
if( OpenProcessToken( GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken ) )
{
    priv.PrivilegeCount           = 1;
    priv.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    if( LookupPrivilegeValue( NULL, SE_DEBUG_NAME, &priv.Privileges[0].Luid ) )
        AdjustTokenPrivileges( hToken, FALSE, &priv, 0, NULL, NULL );

    CloseHandle( hToken );
}
```

بعد از ارتقای سطح دسترسی کنترل را فرایند بدست می‌آوریم با خصوصیاتی که توضیح خواهیم داد.

```
hProcess = OpenProcess( PROCESS_CREATE_THREAD | PROCESS_QUERY_INFORMATION | PROCESS_VM_OPERATION | PROCESS_VM_WRITE | PROCESS_VM_READ, FALSE, dwProcessId );
```

در پارامتر اول مشخص کرده‌ایم که کنترل بدست آمده از فرایند دارای چه ویژگی‌هایی باشد.

PROCESS_CREAT_THREAD : برای ساخت thread جدید در فرایند استفاده می‌شود.

PROCESS_QUERY_INFORMATION : برای بدست آوردن اطلاعات مورد نیاز از فرایند استفاده می‌شود.

PROCESS_VM_OPRATION : برای صدور اجازه دسترسی به آدرس فضای فرایند

PROCESS_VM_WRITE : برای خواندن از حافظه‌ی فرایند

PROCESS_VM_READ : برای نوشتن در حافظه‌ی فرایند

پارامتر بعدی مشخص می‌کند که فرایند باز شده از کنترل فرایند فعلی ارث ببرد یا خیر.

پارامتر بعدی مشخص کننده فرایند هدف برای باز کردن می‌باشد.

بعد اجرا این تابع کنترل به دست آمده از فرایند هدف به وجود آمده و قابل دسترس از طریق متغیر hProcess می‌باشد.

تزریق dll به فرایند هدف

برای تزریق dll در فرایند یک تابع تعریف شده که از آن استفاده می‌کنیم و اطلاعات مورد نیاز را برای استفاده در تابع در اختیار تابع قرار می‌دهیم.

```
hModule = LoadRemoteLibraryR( hProcess, lpBuffer, dwLength, NULL );
```

hProcess کنترلی از فرایند است. lpBuffer کنترلی از dll بارگذاری شده در حافظه است و dwLength حجم dll می‌باشد. پارامتر آخر مربوط به فایل تمپلیت است که مورد استفاده قرار گرفته نشده است و مقدار null می‌باشد.

اولین کاری که داخل تابع انجام می‌شود موقعیت نسبی تابعی از dll که باید فراخوانی شود را بدست می‌آوریم. با فرستادن آدرس dll به تابعی که خود تعریف کرده‌ایم.

```
dwReflectiveLoaderOffset = GetReflectiveLoaderOffset( lpBuffer );
```

محتویات داخل تابع به دلیل دور شدن از بحث گفته نشده اما می‌توانید با مراجعه به پیوست مقاله کد استافده شده داخل تابع را مشاهده کنید.

این تابع با استفاده از خواند اطلاعات داخل هیدر لود شده در حافظه از dll موقعیت نسبی (offset) تابع مورد نظر ما که نامش ReflectiveLoader می‌باشد محاسبه می‌کند و بر می‌گرداند.

بعد از بدست آوردن موقعیت تابع بخشی را برای باگذاری dll در فرایند هدف، به dll اختصاص می‌دهیم.

```
lpRemoteLibraryBuffer = VirtualAllocEx( hProcess, NULL, dwLength, MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE );
```

پارامتر اول کنترل فرایند، پارامتر دوم برای دادن آدرسی خاص استفاده می‌شود که ما از این ویژگی استفاده نمی‌کنیم، پارامتر سوم حجم فضا را مشخص می‌کند، پارامتر چهارم نوع حافظه‌ی تخصیص داده شده است که در اینجا مشخص کرده‌ایم که محدوده‌ی اختصاص داده شده رزرو شود و آماده اعمال تغییرات باشد، به این بخش از حافظه اجازه می‌دهد که اجرا شود، فقط خواندنی باشد یا خواندنی و نوشتی باشد.

در مرحله‌ی بعدی dll بارگذاری شده در حافظه را به بخشی تخصیص داده شده از فرایند هدف منتقل می‌کنیم.

```
WriteProcessMemory( hProcess, lpRemoteLibraryBuffer, lpBuffer, dwLength, NULL )
```

در اینجا مشخص شده که در کنترل فرایند (hProcess) در آدرس حافظه‌ی تخصیص داده شده (lpRemoteLibraryBuffer) اطلاعات مربوط به dll که مورد نظر که قبلا در حافظه بارگذاری شده (lpBuffer) طبق حجم مشخص شده (dwLength) انتقال داده شود.

پارامتر آخر برای استفاده از متغیری است که مشخص می‌کند تعدادی بایت‌های منتقل داده شده در آن دریافت شوند که استفاده از آن اختیاری است و در اینجا از آن استفاده نشده است.

در حال حاضر dll در فرایند هدف بارگذاری شده و تنها یک مرحله‌ی دیگر برای تکمیل عمل dll injection باقی مانده است.

اجرا dll در فرایند هدف

اول باید موقعیت تابع مورد نظر خود را (تابع ReflectiveLoader) که در dll بارگذاری شده در فرایند است پیدا کنیم که خیلی ساده است، کافی است موقعیت نسبی تابع را که در مراحل قبلی بدست آورده‌ایم به آدرس اولین نقطه‌ی dll بارگذاری شده اضافه کنیم که نتیجه موقعیت تابع است.

```
lpReflectiveLoader = (LPTHREAD_START_ROUTINE)( (ULONG_PTR)lpRemoteLibraryBuffer + dwReflectiveLoaderOffset );
```

در مرحله بعد برای اجرا تابع از dll باید یک thread در فرایند هدف به صورت remote ایجاد کنیم و تابع را در این thread جدید اجرا کنیم که این کار با تابع زیر انجام می‌شود.

```
hThread = CreateRemoteThread( hProcess, NULL, 1024*1024, lpReflectiveLoader, lpParameter, (DWORD)NULL, &dwThreadId );
```

پارامتر ها عبارت‌اند از : ۱- کنترل مربوط به فرایند ۲- اشاره‌گری به خصوصیات امنیتی که استفاده نشده است ۳- تخصیص حافظه برای پشته‌ی مربوط به thread ۴- آدرس شروع thread که برابر با آدرس تابع مورد نظر است ۵- اشاره‌گری از متغیری که به تابع فرستاده شود که داینجا lpParameter (برابر NULL) می‌باشد ۶- یک flag که ایجاد thread را کنترل می‌کند ۷- اشاره‌گری به متغیری که مشخصه‌ی thread را می‌باشد.

بعد از اجرای این تابع dll تزریق شده در فرایند هدف اجرا می‌شود و تا پایان عملکرد خود در حافظه‌ی مربوط به فرایند باقی می‌ماند.

در آخر تابع فعلی (LoadRemoteLibraryR) یک کنترل از thread جدید بازگردانده می‌شود که با تابع زیر مشخص می‌کنیم که تا پایان اجرای کامل dll برنامه متوقف شود.

```
WaitForSingleObject( hModule, -1 );
```

البته می‌توان این تابع را حذف کرد و از ادامه برنامه خود جلوگیری نکنیم و مشکلی هم پیش نخواهد آمد فقط اگر فرایند مربوط به همین برنامه را بخواهیم مورد حمله قرار دهیم برای اینکه نتایج قابل مشاهده باشد باید از ادامه فرایند برنامه جلوگیری کرد ولی اگر باز برنامه را متوقف نکنیم thread جدید کار خود را بدون ایراد انجام خواهد داد.

در آخر برنامه نیز برای آزاد کردن حافظه‌ی اضافی باید حافظه استفاده شده را آزاد کنیم.

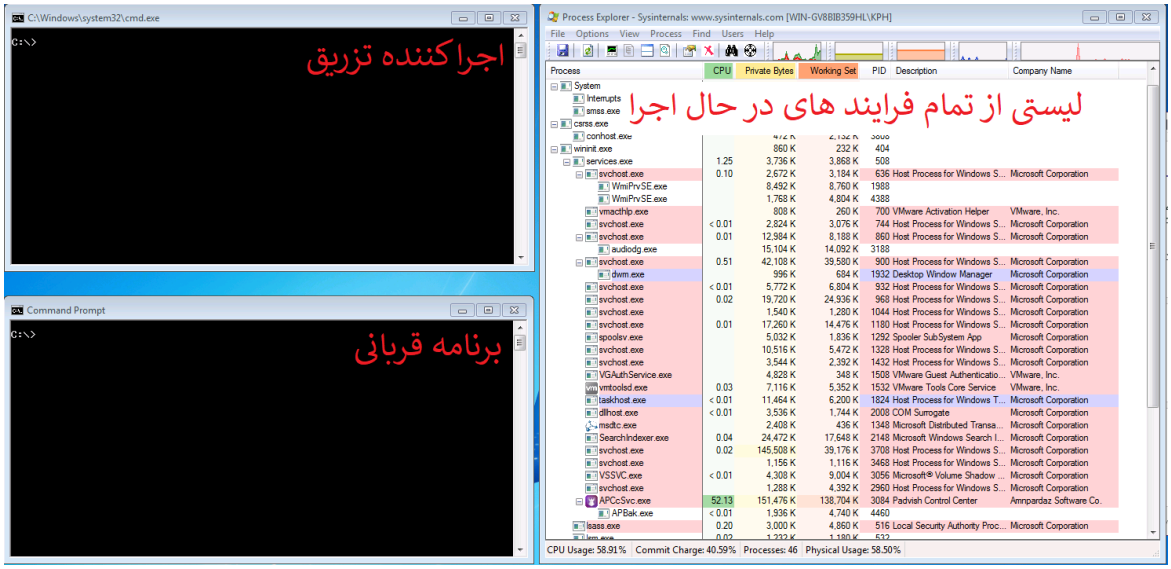
```
if( lpBuffer )
    HeapFree( GetProcessHeap(), 0, lpBuffer );

if( hProcess )
    CloseHandle( hProcess );
```

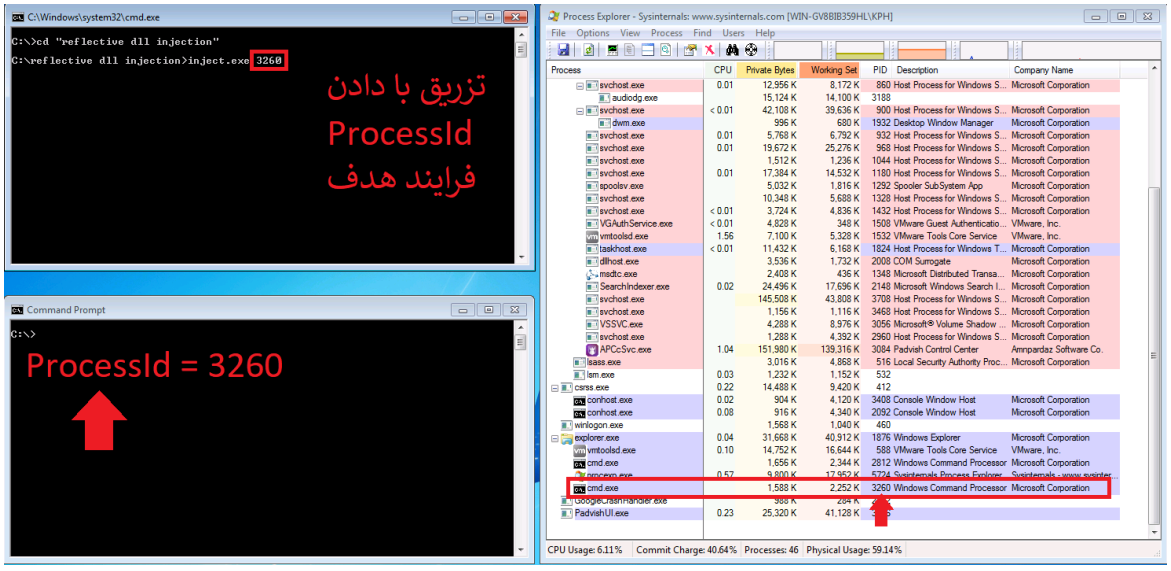
نتایج تجربی

در این بخش آزمایش انجام شده، به صورت تصویری و مرحله به مرحله نشان داده شده است.

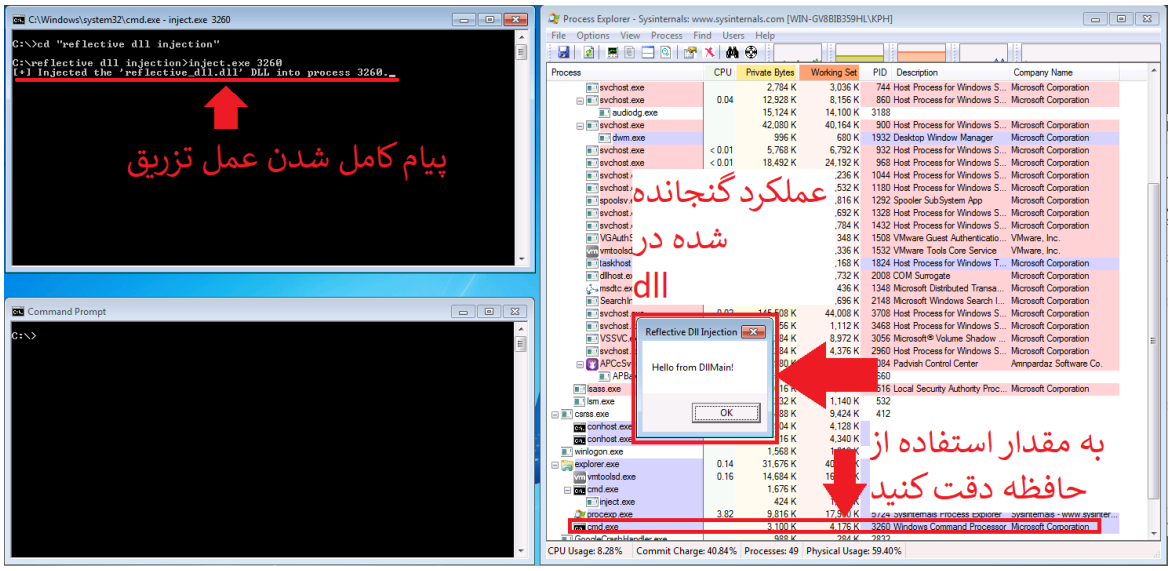
برای تزریق باید اول processid فرایند هدف را شناسایی کنیم که ما در این آزمایش از برنامه Process explorer برای دیافت این اطلاعات استفاده می‌کنیم و فرایند قربانی یا هدف مربوط به برنامه CMD تحت سیستم عامل‌های ویندوز می‌باشد. اول ما دو CMD یک برای انجام تزریق و دیگری همان طور که گفته شد به عنوان قربانی در نظر می‌گیریم.



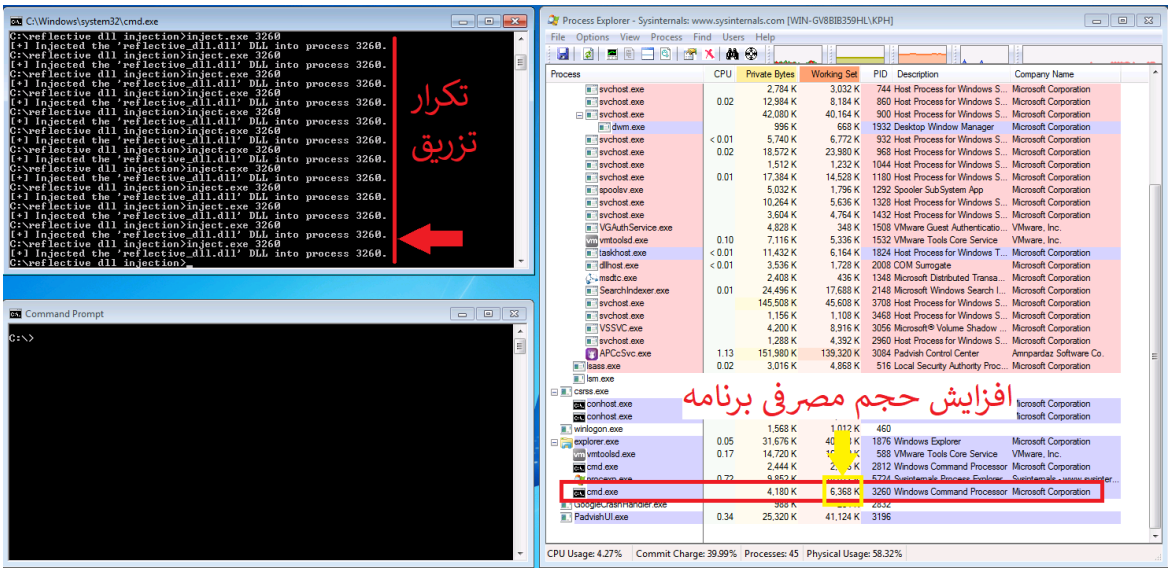
در مرحله‌ی بعد processed مربوط به CMD دوم را با استفاده از Process explorer به دست می‌آوریم و از این id برای اجرای dll injection استفاده می‌کنیم.



حال تزریق dll را انجام می‌دهیم و منتظر بازخورد تزریق dll می‌شویم. که در اینجا این بازخود یک messagebox می‌باشد که حاوی اطلاعاتی است که در dll گنجانده شده است و همینطور پیامی در CMD اول که به معنی با موفقیت انجام شدن تزریق می‌باشد.



اگر به تصویر بالا دقت کنید متوجه می‌شوید که حجم حافظه‌ی مصرفی از سوی فرایند هدف افزایش پیدا کرده است که به معنی وجود dll در حافظه‌ی اختصاص داده شده به این فرایند است برای درک بهتر این تغییر چند بار دیگر این تزریق را انجام می‌دهیم البته این dll حاوی عملکرد مخربی نیست. و تکرار این عمل فقط فضای بیشتری از فرایند را به خود اختصاص می‌دهد که موجب افزایش حجم فرایند می‌شود.



مقدار جدید را با مقادیر قبلی مقایسه کنید، اختلاف چشم‌گیر است. البته این اختلاف به وجود آمده در حجم حافظه برنامه قربانی راه مناسبی برای تشخیص dll injection نمی‌باشد چون برنامه‌های مختلف ممکن است خیلی اختلافات چشمگیر تری در حجم مصرفی حافظه داشته باشند در این آزمایش به این دلیل یک cmd به عنوان قربانی در نظر گرفته شد چود یک cmd دستوری از کاربر دریافت نکند تغییراتی در حافظه خود صورت نمی‌دهد و یا اینکه بسیار ناچیز است و تغییرات این چنینی ملموس تر جلوه خواهند کرد.

نکته : در نشانی microsft process utilities ابزارهای کارآمدی وجود دارند از جمله ProcDump, process explorer, process monitorin و ... که می‌توانند کمک بسیاری در بررسی عملکرد process ها به محققان در مورد بررسی process ها داشته باشند.

ایده‌های قابل اجرا در مورد جلوگیری از Memory Injection

ایده اولیه از سوی شرکت Lumension

شرکت لیومنشن، یک شرکت کامپوتری فعال در حوزه امنیت شبکه، تنها شرکتی است مدعی است که می‌تواند حملات از نوع Reflective Memory Injection(RMI) را می‌تواند تشخیص دهد و از آن جلوگیری کند اما خب مثل هر شرکت نرم‌افزاری دیگر این شرکت نیز روش کار خود را به صورت محرمانه و اختصاصی برای خود به عنوان یک امتیاز نگه داشته است، با این حال توضیح مختصر و کوتاهی از روش کار خود داده‌اند که به شرح زیر می‌باشد.

Aplicaion Control مربوط به محصول این شرکت از سیستم WhiteListing برای جلوگیری از اجرای فایل‌های اجرایی استفاده می‌کند. و همچنین دارای یک قابلیت به نام CoreTrace است که دارای تکنولوژی Bouncer است که می‌تواند با مانیتور کردن آدرس Endpoint ی از حافظه از حملات RMI جلوگیری کند. و همین طور با گسترش WhiteList قادر خواهد بود که از فرایند های اجرایی که از سوی برنامه‌های غیر مجاز اجرا می‌شوند نیز شناسایی کرده و از اجرای آن‌ها جلوگیری می‌کند.

ایده‌های قابل اجرا برای تشخیص تزریق در حافظه

تقریبا در تمام روش‌های پیشنهادی استفاده از تکنیک WhiteList اثر بسیار زیادی در سرعت و کارآمدی روش‌های پیشنهادی دارد.

- خواندن اطلاعات مربوط به استک هر فرایند و یافتن آدرس‌های بازگشتی که متعلق به یک فرایند مجاز نیستند و به نوعی دسترسی خارجی محسوب می‌شود.
  این روش پیشنهادی به نظر درصد موفقیت بسیار پایینی دارد چون تشخیص آدرس از روی استک یک فرایند وابسته به اطلاعاتی است که به سرعت در حال تغییر هستند و احتمال شکست در شناسایی فرایند مهاجم بسیار بالا است در مخصوصا در صورتی که فرایند مهاجم از تکنیک های دفایی مانند حذف خود از روی حافظه بعد از اجرا شدن خود استفاده کند که در این صورت آدرس های موجود درون استک خود را نیز پاک خواهد کرد برنامه با این تکنیک غیر قابل شناسایی می‌شود.
- کنترل سطح دسترسی داده شده به بخش‌های مختلف از حافظه‌ی فرایندهایی معین و ثبت تغییرات در این خصیصه.
  در این روش باید یک Event Handler بر روی بخش های مختلف حافظه قرار داد و بررسی کرد که اگر سطح دسترسی بخشی از حافظه اگر به RWX تغییر کرد یک رویداد رخ دهد و بررسی کند که این تغییر آیا از سوی یک برنامه مجاز بوده یا یک برنامه نامشخص و غیرجاز که در این صورت با جلوگیری از اعمال تغییرات و بازگردانی سطح دسترسی

حافظه به مقدار اولیه و متوقف کردن برنامه مهاجم از دسترسی های غیر مجاز در حافظه جلوگیری می‌شود. البته با وجود مزایای این روش از جمله سرعت بالا در اجرا و کمتر هزینه داشتن برای سیستم اما به راحتی قابل دور زدن است.

- نظارت بر استفاده از APIهای حساس که در این گونه حملات مورد استفاده قرار می‌گیرند و یافتن فرایندهای قربانی شده و مهاجم با این روش.

برای استفاده از این روش اول باید توانست API هایی که احتمال استفاده در تزیرق کد را دارند یافت که این خود نیازمند تحقیقات گسترده‌ای در مورد API های دارای آسیب پذیری می‌باشد. بعد از شناسایی این API های حساس با مانیتور کردن استفاده هایی که از این API ها می‌شود می‌توان حملات احتمالی را تشخیص داد. برای دقیق‌تر شدن در تشخیص حملات بهتر است یک WhiteList از فرایندهای دارای حساسیت تهیه کرد و در هنگام مانیتور کردن اطلاعات API های استفاده شده به این دقت داشت که برنامه‌ی غیرمجازی به این به برنامه های موجود در WhiteList دسترسی پیدا نکند و در حافظه این برنامه‌ها تغییری ایجاد نکند. که در غیر این صورت برنامه‌ای که مهاجم شناخته شده است متوقف شود.

- ثبت dllهای مجاز برای هر فرایند و بررسی استفاده و وجود دیگر dllها در فرایندهایی که مورد بررسی قرار گرفته‌اند.

ساده‌ترین راهی که می‌توان پیش گرفت این است که یک WhiteList برای هر یک فرایندهای موجود در سیستم تهیه کرد و بررسی کرد که اگر dllی دیگر به فرایند اضافه شد برنامه متوقف شده و درخواست اجازه از کاربر کند یا اینکه برنامه را متوقف کند تا از نفوذ مهاجم در برنامه جلوگیری کند. این روش دفاعی توسط برخی توسعه دهندگان برنامه‌های کاربردی به صورت درون ساخت در برنامه خود قرار می‌دهند تا از آلوده شدن خود جلوگیری کنند.

- گرفتن Hash از بخش‌های مختلف از حافظه‌ی یک فرایند و ذخیره این مقادیر و بررسی این مقادیر در هنگام اجرا با مقادیر ثبت شده‌ی قبلی برای یافتن تغییرات غیرمجاز احتمالی.

این روش بسیار مؤثر در پیدا کردن تغییرات در بخش‌های مختلف حافظه است و این کار به سادگی با گرفتن hash از بخش های مختلف یک فرایند و ذخیره این مقادیر است و در مرحله‌ی بعد در موقع اجرا فقط کافی است که با hash گیری دوباره از بخش‌های حافظه و مقایسه با مقادیر قبلی تغییرات در حافظه فرایندهای مد نظر را تشخیص داد. ولی با این حال این روش دارای درصد false-positive بالایی خواهد بود.

فایل‌های پیوست

سورس کد تابع RVA2Offset

```
DWORD Rva2Offset( DWORD dwRva, UINT_PTR uiBaseAddress )
{
    WORD wIndex                         = 0;
    PIMAGE_SECTION_HEADER pSectionHeader = NULL;
    PIMAGE_NT_HEADERS pNtHeaders         = NULL;

    pNtHeaders = (PIMAGE_NT_HEADERS)(uiBaseAddress + ((PIMAGE_DOS_HEADER)uiBaseAddress)->e_lfanew);

    pSectionHeader = (PIMAGE_SECTION_HEADER)((UINT_PTR)(&pNtHeaders->OptionalHeader) + pNtHeaders->FileHeader.SizeOfOptionalHeader);

    if( dwRva < pSectionHeader[0].PointerToRawData )
        return dwRva;

    for( wIndex=0 ; wIndex < pNtHeaders->FileHeader.NumberOfSections ; wIndex++ )
    {
        if( dwRva >= pSectionHeader[wIndex].VirtualAddress &&
            dwRva < (pSectionHeader[wIndex].VirtualAddress + pSectionHeader[wIndex].SizeOfRawData) )
          return ( dwRva - pSectionHeader[wIndex].VirtualAddress + pSectionHeader[wIndex].PointerToRawData );
    }

    return 0;
}
```

سورس کد تابع GetReflectiveLoaderOffset

```
DWORD GetReflectiveLoaderOffset( VOID * lpReflectiveDllBuffer )
{
    UINT_PTR uiBaseAddress   = 0;
    UINT_PTR uiExportDir     = 0;
    UINT_PTR uiNameArray     = 0;
    UINT_PTR uiAddressArray  = 0;
    UINT_PTR uiNameOrdinals  = 0;
    DWORD dwCounter          = 0;
#ifdef WIN_X64
    DWORD dwCompiledArch = 2;
#else
    DWORD dwCompiledArch = 1;// This will catch Win32 and WinRT.
#endif
    uiBaseAddress = (UINT_PTR)lpReflectiveDllBuffer;
    uiExportDir = uiBaseAddress + ((PIMAGE_DOS_HEADER)uiBaseAddress)->e_lfanew;// get the File Offset of the modules NT Header
    // currenlty we can only process a PE file which is the same type as the one this fuction has
    // been compiled as, due to various offset in the PE structures being defined at compile time.
    if( ((PIMAGE_NT_HEADERS)uiExportDir)->OptionalHeader.Magic == 0x010B ){ // PE32
        if( dwCompiledArch != 1 )return 0;
    }
    else if( ((PIMAGE_NT_HEADERS)uiExportDir)->OptionalHeader.Magic == 0x020B ){ // PE64
        if( dwCompiledArch != 2 )return 0;
    }
    else{return 0;}
    // uiNameArray = the address of the modules export directory entry
    uiNameArray = (UINT_PTR)&((PIMAGE_NT_HEADERS)uiExportDir)->OptionalHeader.DataDirectory[ IMAGE_DIRECTORY_ENTRY_EXPORT ];
    // get the File Offset of the export directory
    uiExportDir = uiBaseAddress + Rva2Offset( ((PIMAGE_DATA_DIRECTORY)uiNameArray)->VirtualAddress, uiBaseAddress );
    // get the File Offset for the array of name pointers
    uiNameArray = uiBaseAddress + Rva2Offset( ((PIMAGE_EXPORT_DIRECTORY )uiExportDir)->AddressOfNames, uiBaseAddress );
    // get the File Offset for the array of addresses
    uiAddressArray = uiBaseAddress + Rva2Offset( ((PIMAGE_EXPORT_DIRECTORY )uiExportDir)->AddressOfFunctions, uiBaseAddress );
    // get the File Offset for the array of name ordinals
    uiNameOrdinals = uiBaseAddress + Rva2Offset( ((PIMAGE_EXPORT_DIRECTORY )uiExportDir)->AddressOfNameOrdinals, uiBaseAddress );
    // get a counter for the number of exported functions...
    dwCounter = ((PIMAGE_EXPORT_DIRECTORY )uiExportDir)->NumberOfNames;
    // loop through all the exported functions to find the ReflectiveLoader
    while( dwCounter-- )
    {
        char * cpExportedFunctionName = (char *)(uiBaseAddress + Rva2Offset( DEREF_32( uiNameArray ), uiBaseAddress ));
        printf("%s\n", cpExportedFunctionName);
        if( strstr( cpExportedFunctionName, "ReflectiveLoader" ) != NULL )
        {
            // get the File Offset for the array of addresses
            uiAddressArray = uiBaseAddress + Rva2Offset( ((PIMAGE_EXPORT_DIRECTORY )uiExportDir)->AddressOfFunctions, uiBaseAddress );

            // use the functions name ordinal as an index into the array of name pointers
            uiAddressArray += ( DEREF_16( uiNameOrdinals ) * sizeof(DWORD) );

            // return the File Offset to the ReflectiveLoader() functions code...
            return Rva2Offset( DEREF_32( uiAddressArray ), uiBaseAddress );
        }
        // get the next exported function name
        uiNameArray += sizeof(DWORD);

        // get the next exported function name ordinal
        uiNameOrdinals += sizeof(WORD);
    }

    return 0;
}
```

منابع

[1] Antoniewicz, B. (2013, January 8). *Windows DLL Injection Basics*. Retrieved from Open Security Research: http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html

[2] Microsoft ,Ds. (n.d.). *Sysinternals Process Utilities*. Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/sysinternals/downloads/process-utilities

[3] Fewer, S. (2008, October 31). *Reflective DLL Injection.* Retrieved from Harmony Security.

[4] Fewer, S. (2013, September 5). *Reflective DLL Injection*. Retrieved from github: https://github.com/stephenfewer/ReflectiveDLLInjection

[5] Korkin, I. (22 may 2017). DETECT KERNEL-MODE ROOTKITS VIA REAL TIME LOGGING & CONTROLLING MEMORY ACCESS. *The 12th ADFSL Conference on Digital Forensics, Security and Law*, (p. 31). Moscow, Russia.

[6] Microsoft, M. (n.d.). *PE Format*. Retrieved from MSDN Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx

[7] Microsoft, M. (n.d.). *Traversing the Module List*. Retrieved from MSDN Microsoft: https://msdn.microsoft.com/en-us/library/windows/desktop/ms686849(v=vs.85).aspx

[8] MSDN, M. (n.d.). *Memory Management Functions*. Retrieved from Microsoft Msdn : https://msdn.microsoft.com/en-us/library/windows/desktop/aa366781(v=vs.85).aspx

[9] Skape, J. T. (2004, April 6). *Remote Library Injection*. Retrieved from No Login.