اصول shellcode نویسی

۱. مقدمه

یک shellcode مجموعه دستوری است که می تواند در خلل اجرای برنامه ای دیگر اجرا گردد. امروزه بسیاری از نمونه ها چگونگی اجرای یک shellcode را در خلال اجرای یک برنامه نشان داده اند. معمولا گام بعدی استفاده از shellcode ها، کدهای اکسپلویتی است که برای سواستفاده از آسیب پذیری ها نوشته می شود. برای اینکه بتوان از یک آسیب پذیری استفاده کرد (در جهت سوا!)، تزریق یک shellcode ضروری و حتمی است، چرا که ما مجبور به در اختیار گرفتن کنترل برنامه در حال اجرا هستیم.

هدف این مقاله این نیست که تمام احتمالات موجود در تزریق یک shellcode را توضیح دهد، بلکه هدف آنالیز و درک ضرورت این پدیده یعنی shellcode می باشد.

۲. ثبات ها

قبل از آنالیز کدهای اسمبلی و متعاقبا نمونه باینری از کد، مروری سریع بر وضعیت ثبات های ریز پردازنده (جهت درک اهمیت آنها در زبان اسمبلی) ضروری بنظر می رسد.

ساختارهای معماری که در این مقاله مورد بررسی قرار می دهیم، Intel-x86 خواهد بود. تمامی ثبات های پلاتفرم Intel از اندازه های ۳۲ بیتی پشتیبانی می کنند که می توان آنها را به خرده بخش های ۱۶ یا ۸ بیتی تقسیم کرد تا به این صورت استفاده یک شی از حافظه را مقدور سازیم.

32 Bit	16 Bit	8 Bit (high)	8 Bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

جدول ۱: اندازه ثبات های کامپیوتری

EAX، AX، AH، AH: به این ثبات ها انباره یا **accumulator** گفته می شود و می توان آنها را برای عملیات حسابی یا ورودی/خروجی یا اجرای فراخوانی های منقطع استفاده کرد. همان طور که پی بردید، هنگامی که مجبور باشیم فراخوانی های سیستمی را تشخیص دهیم، استفاده از آنها ضروری خواهد بود.

EBX، BX، BH، BL: این ثبات ها، ثبات های پایه هستند و از آنها به عنوان اشاره گرهای پایه در دستیابی به حافظه استفاده می شود. ما از این ثبات ها جهت انتقال آرگومان های فراخوانی های سیستمی استفاده می کنیم. همچنین از آنها جهت ذخیره مقدار برگشتی از یک انقطاع (interrupt) استفاده می شود (برای مثال، هنگامی که open() را فراخوانی می کنیم، مقدار توصیفگر[1] مربوط به le_ در ثبات EBX ذخیره می شود).

[1] Descriptor's value

ECX، CX، CH، CL: به این ثبات ها، ثبات های شمارنده (یا شمارشی) یا **Counter** می گویند.

EDX، DX، DH، DL: این ثبات ها، ثبات های داده ای هستند که میتوان آنها را برای عملیات حسابی، فراخوانی های منقطع و بعضی از عملگرهای ورودی/خروجی مورد استفاده قرار داد.

۳. آشنایی اجمالی با زبان اسمبلی

زبان اسمبلی که ما قصد معرفی آنرا داریم "Inline Assembly" نامیده می شود که از ساختار AT&T اقتباس شده است. نام ثبات ها با علامت "%" ممیز شده است. بنابراین، اگر مجبور به استفاده از ثبات EAX باشیم، باید عبارت "%eax" را تایپ کنیم. اگر بخواهیم ثابت های عددی را ارجاع دهیم، مقدار آن بایستی با علامت "$" همراه باشد. در زیر می توانید پرکاربردترین دستورها در زبان اسمبلی را مشاهده کنید:

MOV: این دستور امکان انتقال یک مقدار (value) را به یک ثبات فراهم می سازد. مثال:

دستور mov $0x4, %al، 0x4 را به al انتقال می دهد.

دستور mov %eax, %ebx، محتویات موجود در EAX را به EBX می برد.

PUSH: این دستور یک مقدار را در پشته (stack) قرار می دهد.

POP: این دستور یک مقدار را از پشته برداشته و آنرا در یک ثبات یا یک متغیر ذخیره می کند.

INT: این دستور، سبب قطع فراخوانی یا انجام عملیات Call Interruption می شود. مثال:

دستور int $0x80، کنترل را به هسته بر می گرداند.

۴. ورود به فاز کدنویسی

الگوریتمی که ما در این بخش در زبان اسمبلی و پس از آن در کد باینری (به عنوان نسخه هگزادسیمال از کد) پیاده سازی خواهیم کرد، چاپ کردن عبارت WWW.CROUZ.COM می باشد. کد مربوط به این برنامه را در زبان C در زیر میبینید:

```
int main()
{
write(0, "WWW.CROUZ.COM", 16);
exit(0);
}
```

برای اینکه write() و exit() را بدرستی درک کنیم، مجبور به اجرای فراخوانی های سیستمی آنها هستیم. این امکان وجود دارد که به جای _nd در لینوکس، شخص از کتابخانه "unistd.h" استفاده کند که تمام فراخوانی های سیستمی در آن موجود است.

```
ceph@terminator$ cat /usr/include/asm-i386/unistd.h

#de_ne _NR_exit 1
#de_ne _NR_fork 2
#de_ne _NR_read 3
#de_ne _NR_write 4
#de_ne _NR_open 5
write(0, "WWW.CROUZ.COM", 16);
```

(خط اول exit() و خط چهارم write() می باشد).

آرگومان rst_ به صورت "0"، خروجی استاندارد می باشد که ما رشته را در آن به عنوان آرگومان ثانویه چاپ میکنیم. آخرین آرگومان یعنی "16" طول رشته را مشخص می کند. اکنون بیایید این دستور را در اسمبلی پیاده سازی کنیم:

```
xor %eax, %eax
xor %ebx, %ebx
xor %edx, %edx
push %eax
push $0x47524f2e # MOC.
push $0x4f4c4c45 # Z into
push $0x49534f52 # CRO
push $0x2e575757 # .WWW
```

(دستور اول، ثبات eax% را خالی می کند. دستور چهارم، NULL را به پشته اضافه می کند که نشانه خاتمه رشته می باشد (کاراکتر پوچ یا NULL نشانه پایان رشته می باشد)، لذا هیچ کاراکتر اضافی نمایش نمی یابد).

چهار دستور PUSH در انتها، رشته "WWW.CROUZ.COM" را با معادل هگزادسیمال آن، در پشته اضافه می کنند. همان طور که دیدید، رشته باید به صورت برعکس در پشته جای گیرد، چون استراتژی کاری پشته به این گونه است (مدل LIFO را که به یاد می آورید). توصیفگر خروجی استاندارد با ثبات ebx% مرتبط می باشد که فقط حاوی مقدار 0 میباشد، بدینسان مجبور نیستیم که چیزی را در این ثبات تعیین کنیم (بهمین منظور است که از دستور write(0,…) استفاده کرده ایم).

دستور mov %esp, %ecx، محتویات esp% را به ecx% انتقال می دهد. اکنون آدرس رشته در ثبات esp% می باشد (توجه کنید که ESP تنها با pop/push کم و زیاد می شود) و ما آنرا در ثبات ecx% قرار می دهیم، بنابراین پردازنده قادر خواهد بود که مکان صحیح رشته را در پشته (_nd) تشخیص دهد (write(0,string, …).

دستور mov $0x10,%dl ۱۶ بایت را درگیر کار می کند. صراحتا در زبان C می توان نشان داد که طول رشته برابر ۱۶ بایت می باشد ((write(0, string, 16).

دستور mov $0x4,%al، یک فراخوانی سیستمی برای write() می باشد. ما تعداد روتین write() را در ثبات EAX قرار می دهیم (در قسمت پائینی: al).

دستور int $0x80 باعث اجرای فراخوانی سیستمی می شود. اکنون هسته برنامه را بدست گرفته و ما روتین write() خود را اجرا خواهیم کرد. پیاده سازی exit(0) بسیار راحت تر می باشد. در زیر این مورد را می بینید:

```
xor %eax, %eax
xor %ebx, %ebx
```

ثبات های eax و ebx خالی هستند.

دستور mov $0x1, %al، یک فراخوانی سیستمی برای exit() می باشد. در زیر می بینید که ما مقدار مربوط به exit را در al الحاق میکنیم و پس از آن کنترل را به هسته می دهیم:

```
mov $0x1, %al
int $0x80 #execute the syscall
```

۵. کامپایل و اجرا

آخرین گام باید کدنویسی های انجام شده را به کدهای باینری تبدیل کنیم. به این منظور از GDB استفاده خواهیم کرد:

```
ceph@terminator:\_shellcode$ gdb ceph

(gdb) disas main
Dump of assembler code for function main:
0x80482f4 <main>: push %ebp
0x80482f5 <main+1>: mov %esp,%ebp
0x80482f7 <main+3>: sub $0x8,%esp
0x80482fa <main+6>: and $0x___f0,%esp
0x80482fd <main+9>: mov $0x0,%eax
0x8048302 <main+14>: sub %eax,%esp
0x8048304 <main+16>: xor %eax,%eax
0x8048306 <main+18>: xor %ebx,%ebx
0x8048308 <main+20>: xor %edx,%edx
0x804830a <main+22>: push %eax
0x804830b <main+23>: push $0x47524f2e
0x8048310 <main+28>: push $0x4f4c4c45
0x8048315 <main+33>: push $0x49534f52
0x804831a <main+38>: push $0x2e575757
0x804831f <main+43>: mov %esp,%ecx
0x8048321 <main+45>: mov $0x10,%dl
0x8048323 <main+47>: mov $0x4,%al
0x8048325 <main+49>: int $0x80
0x8048327 <main+51>: xor %eax,%eax
0x8048329 <main+53>: xor %ebx,%ebx
0x804832b <main+55>: mov $0x1,%al
0x804832d <main+57>: int $0x80
End of assembler dump.
```

کد ما از دستور <main+16> شروع شده و در <main+57> پایان می یابد. برای رسیدن به opcode باید مراحل زیر را طی کنیم:

```
(gdb) x/bx main+16
0x8048304 <main+16>: 0x31 <_ OPCODE
(gdb)
0x8048305 <main+17>: 0xc0 <_ OPCODE
(gdb)
0x8048306 <main+18>: 0x31 <_ OPCODE
....
```

و بهمین صورت تا main+57 این عمل را تکرار می کنیم. اکنون باید همه چیز را با الگوی \x (مثلا: "\x31\xc0\x31...") تنظیم کنیم، همان طور که در زیر می بینید:

```
"\x31\xc0\x31\xdb\x31\xd2\x50\x68\x2e\x4f"
"\x52\x47\x68\x45\x4c\x4c\x4f\x68\x52\x4f"
"\x53\x49\x68\x57\x57\x57\x2e\x89\xe1\xb2"
"\x10\xb0\x04\xcd\x80\x31\xc0\x31\xdb\xb0"
```

"\x01\xcd\x80"

برای کامپایل و اجرای shellcode، می توانید آنرا بعنوان یک برنامه C مانند طرح زیر سازمان دهی کنید.

```
ceph@terminator:\_shellcode$ cat shellcode.c
#include <stdio.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xd2\x50\x68\x2e\x4f"
"\x52\x47\x68\x45\x4c\x4c\x4f\x68\x52\x4f"
"\x53\x49\x68\x57\x57\x57\x2e\x89\xe1\xb2"
"\x10\xb0\x04\xcd\x80\x31\xc0\x31\xdb\xb0"
"\x01\xcd\x80";

main()
{
void (*routine) ();
(long) routine = &shellcode;
printf("Size: %d bytes\n", sizeof(shellcode));
routine();
}
ceph@terminator:\_shellcode$ gcc shellcode.c -o shellcode
ceph@terminator:\_shellcode$ ./shellcode
Size: 44 bytes.
WWW.CROUZ.COM
```

نتیجه

shellcode نویسی در برنامه های سطح پائین بسیار مهم است. برای مثال، اگر می خواهید اکسپلویتی را کدنویسی کنید، به نوشتن shellcode نیز احتیاج خواهید داشت. چرا که با shellcode، برنامه اکسپلویت شده، کدهای دلخواه ما را اجرا می کند. به هر حال، این مفاهیم پایه و تئوری ها برای شخصی که در علم امنیت کامپیوتر فعالیت می کند حداقل اطلاعات ممکن خواهد بود که منجر به کاوش bug ها و روش های اکسپلویت جدید می شود.

ترجمه: سعید بیکی (cephexin@secumania.net)

© Secumania Security & Vulnerability Research Lab
www.secumania.net