بسم الله الرحمن الرحیم

مقدمه ای بر حملات
Buffer Overflow No.1
((در لینوکس))

مقدمه:

حملات Buffer Overflow یکی از رایج ترین نوع حملات علیه برنامه های آسیب پذیری می باشد.در این حمله با دستکاری مقدار یک متغیر و یا وارد کردن مقداری غیر معتبر و بیش از اندازه به یک متغیر ، برنامه دچار اشکال شده و می توان از این اشکال پیش آمده برای اجرای دستورات مورد نظر استفاده کرد.در این مقاله در سطح مقدماتی با نمونه ای از این حملات در سیستم عامل لینوکس آشنا می شویم.لازم به ذکر است که می توان برنامه ی مثال استفاده شده در این مقاله را در سیستم عامل ویندوز نیز استفاده کرد.لازم به ذکر است که این مقاله تنها یک پیش زمینه برای آشنایی با نوشتن اکسپلویت ها و یافتن آسیب پذیری ها بوده و نمی توان آن را به صورت یک مقاله ی کامل در این زمینه در نظر گرفت.

Exploit چیست؟

شاید اکثر شما بدانید که اکسپلویت چیست ، اما اگر نمی دانید 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.بیشتر اکسپلویت ها از آسیب پذیری های Stack Overflow یا Buffer Overflow استفاده می کنند.لازم به ذکر است که نحوه ی یافتن آسیب پذیری های گفته شده در برنامه های مختلف ، با یکدیگر تفاوت دارند.
می توان با رفتن به سایت های مختلفی مانند www.securityfocus.com اکسپلویت های زیادی را یافت.شما به راحتی می توانید با دریافت سورس این اکسپلویت ها و تبدیل آن ها به فرمت اجرایی ، از آسیب پذیری که اکسپلویت برای آن نوشته شده است ، استفاده کنید.اما چرا هر فرد نمی تواند یک اکسپلویت بنویسد؟مساله ی اصلی این است که هر فرد نمی تواند نقطه ی آسیب پذیری را در برنامه ای پیدا کند ، یا حتی اگر آسیب پذیری پیدا کند ، بدون داشتن اطلاعات و روش های اکسپلویت نویسی نمی تواند یک اکسپلویت بنویسد.بهترین ایده برای اکسپلویت نویسی استفاده از آسیب پذیری یافته شده است که شاید بزرگترین تهدید برای اینترنت باشد.

Buffer Overflow چیست؟

همان طور که گفته شد بیشتر آسیب پذیری های یافت شده از نوع Buffer Overflow هستند اما حال Buffer Overflow چیست؟آسیب پذیری Buffer Overflow در قسمتی از حافظه که اطلاعات و مقادیر متغیرها و رشته ها را در خود نگه می دارد و Buffer نامیده می شود ، اتفاق

می افتد.به این صورت که اگر بیش از اندازه ی Buffer در آن بنویسیم ، به دلیل کمبود جا برای نوشتن سایر اطلاعات ، به حالتی به نام Overflow دچار می شود.یعنی نوشتن زیاد از حد اطلاعات در قسمت خاصی از Buffer سیستم.برای درک بهتر به مثال زیر توجه کنید:

```
/*Example Code No.2*/
void function1(char *str)
{
  char buffer[10];
  strcpy(buffer,str);
}

void main()
{
  char string1[256];
  int i;
  for(i=0;i<255;i++)
    string1[i]='A';
  function1(string1);
}
```

در این مثال ، ابتدا رشته ای با طول 10 در تابع function1 ایجاد می کنیم و سپس با استفاده از دستور strcpy ، مقدار آرگومان دریافت شده یعنی اشاره گر str را در داخل buffer کپی می کنیم.اما آیا اندازه ی این دو با یکدیگر برابر هستند؟بیشترین آسیب پذیری ها از چک نکردن طول دو آرایه یا رشته ناشی می شوند.در تابع اصلی برنامه رشته ی string1 را با طول 256 تعریف می کنیم.متغیر I را نیز برای شمارش دفعات تکرار حلقه ی تکرار تعریف می کنیم.در یک حلقه ی تکرار ، به هر خانه از رشته ی string1 یک کاراکتر 'A' می دهیم.حال تابع function1 را اجرا کرده و می بینیم که برنامه دچار مشکل می شود:

```
[root@localhost] ./bufferoverfflow1
Segmentation fault (Core Dumped)
[root@localhost]
```

برای درک بیشتر مثال دیگری را می بینیم :

```
/*Example Code No.1*/
void main(int arg
c, char *argv)
{
  char *somevar;
  char *important;
  somevar = (char *)malloc(sizeof(char)*4);
  important = (char *)malloc(sizeof(char)*14);
  strcpy(important, "command");/*This one is the important
variable*/
  stcrpy(somevar, argv[1]);
}
```

اگر متغیر important را به عنوان متغیری که دستورات سیستمی را در خود دارد در نظر بیگیریم ، و به عنوان مثال دستور chmod o-f file را در خود ذخیره کرده است ، و زمانی که برنامه تحت کاربر root اجرا می شود ، این بدان معنا است که شما می توانید هر دستور سیستمی را به برنامه فرستاده و اجرا کنید.حال ما چگونه می توانیم در هنگام اجرای برنامه در متغیر مقدار دلخواهی را که می خواهیم بگذاریم.تنها راه رسیدن به این کار ، Overflow کردن حافظه

است.اما اول برای این کار باید آدرس متغیرها را داشته باشیم. برای دیدن این آدرس ها ،کد بالا را با کمی تغییر دوباره می نویسیم :

```
main (int argc, char *argv)
{
   char *somevar;
   char *important;
   somevar=(char *)malloc(sizeof(char)*4);
   important=(char *)malloc(sizeof(char)*14);
   printf("%p\n%p", somevar, important);
   exit(0);
}
```

ما دو خط به سورس برنامه اضافه کردیم.حال ببینیم این دو خط چه کاری را انجام می دهند.خط اول یعنی printf("%p\n%p", somevar, important) آدرس متغیرهای مورد نظر را چاپ می کند.خط دوم یعنی دستور exit(0) از برنامه خارج می شود.نبود این دستور اشکالی در اجرای برنامه به وجود نمی آورد.بعد از اجرای برنامه شما باید خروجی شبیه زیر دریافت کنید البته شاید آدرس دریافتی شما متفاوت باشد :

```
[root@localhost] ./bufferoverfflow2
0x8049700  <----- This is the address of somevar
0x8049710  <----- This is the address of important
[root@localhost]
```

همان طور که می بینیم ، متغیر important در جایگاه دوم و بعد از somevar قرار دارد ، این عامل اجازه ی Overflow کردن حافظه ر ا به ما می دهد.، البته از زمانی که متغیر somevar از طریق argv[1] مقدار می گیرد.حال ما می دانیم که دو متغیر پشت سر یکدیگر قرار دارند ، اما چک کردن آدرس های به دست آمده تصویر دقیق تری را در مورد اطلاعات ذخیره شده به ما می دهد.حال دوباره کد را برای دریافت اطلاعات بیشتر باز نویسی می کنیم :

```
main(int argc, char *argv)
{
char *somevar;
char *important;
char *temp; /* will need another variable */
somevar=(char *)malloc(sizeof(char)*4);
important=(char *)malloc(sizeof(char)*14);
strcpy(important, "command");  /*This one is the important
variable*/
stcrpy(str, argv[1]);
printf("%p\n%p\n", somevar, important);
printf("Starting To Print memory address:\n");
temp = somevar; /* this will put temp at the first memory address we want

*/

while(temp < important + 14) {

/* this loop will be broken when we get to the last memory address we
```

```
want, last memory address of important variable */
printf("%p: %c (0x%x)\n", temp, *temp, *(unsigned int*)temp);
temp++;
}
exit(0);
}
```

حال برای argv[1] یک دستور معمولی در نظر می گیریم.شما تنها لازم است که این دستور را در خط فرمان خود بنویسید:

```
[root@localhost]./bufferoverflow3 send
```

و خروجی زیر را با ممکن است با آدرس های متفاوتی دریافت کنید :

```
[root@localhost]./bufferoverflow3 send

0x8049700

0x8049710

Starting To Print memory address:

0x8049700: s (0x616c62)
0x8049701: e (0x616c)
0x8049702: n (0x61)  <---- each of this lines represent a memory address
0x8049703: d (0x0)
0x8049704:   (0x0)
0x8049705:   (0x0)
0x8049706:   (0x0)
0x8049707:   (0x0)
0x8049708:   (0x0)
0x8049709:   (0x19000000)
0x804970a:   (0x190000)
0x804970b:   (0x1900)
0x804970c:   (0x19)
0x804970d:   (0x63000000)
0x804970e:   (0x6f630000)
0x804970f:   (0x6d6f6300)
0x8049710: c (0x6d6d6f63)
0x8049711: o (0x616d6d6f)
0x8049712: m (0x6e616d6d)
0x8049713: m (0x646e616d)
0x8049714: a (0x646e61)
0x8049715: n (0x646e)
0x8049716: d (0x64)
0x8049717:   (0x0)
0x8049718:   (0x0)
0x8049719:   (0x0)
0x804971a:   (0x0)
0x804971b:   (0x0)
0x804971c:   (0x0)
0x804971d:   (0x0)

[root@localhost]
```

شما می توانید 12 آدرس خالی را بین آدرس های متغیرهای important و somevar را ببینید.این آدرس های خالی را می توان با مقادیر دیگری پر کرد.دستوری شبیه به دستوری زیر در خط فرمان خود بنویسید و برنامه را اجرا کنید :

[root@localhost]./bufferoverflow3 send----------newcommand

حال خروجی زیر را با آدرس های متفاوت خواهید دید:

```
0x8049700

0x8049710

Starting To Print memory address:

0x8049700: s (0x646e6573)
0x8049701: e (0x2d646e65)
0x8049702: n (0x2d2d646e)
0x8049703: d (0x2d2d2d64)
0x8049704: - (0x2d2d2d2d)
0x8049705: - (0x2d2d2d2d)
0x8049706: - (0x2d2d2d2d)
0x8049707: - (0x2d2d2d2d)
0x8049708: - (0x2d2d2d2d)
0x8049709: - (0x2d2d2d2d)
0x804970a: - (0x2d2d2d2d)
0x804970b: - (0x2d2d2d2d)
0x804970c: - (0x2d2d2d2d)
0x804970d: - (0x6e2d2d2d)
0x804970e: - (0x656e2d2d)
0x804970f: - (0x77656e2d)
0x8049710: n (0x6377656e) <--- memory address where important variable starts
0x8049711: e (0x6f637765)
0x8049712: w (0x6d6f6377)
0x8049713: c (0x6d6d6f63)
0x8049714: o (0x616d6d6f)
0x8049715: m (0x6e616d6d)
0x8049716: m (0x646e616d)
0x8049717: a (0x646e61)
0x8049718: n (0x646e)
0x8049719: d (0x64)
0x804971a:   (0x0)
0x804971b:   (0x0)
0x804971c:   (0x0)
0x804971d:   (0x0)
```

آدرس های خالی حافظه با مقادیر داده شده پر گردیدند اما اگر همه ی خانه های خالی حافظه پر شوند ، می توان این اطمینان را داد که برنامه درست اجرا خواهد شد؟حال در مورد این که در این برنامه چه اتفاقی روی داد ، بحث می کنیم.متغیر somevar قبل از متغیر important تعریف شده است.این متغیر در حافظه اول خواهد بود.حال ببینیم که هر کدام از متغیرها چه مقادیری را گرفته اند.متغیر Somevar مقدار خود را از argv[1] می گیرد ، متغیر important نیز مقدار خود را از طریق تابع strcpy() می گیرد ، اما مساله ی اصلی این جاست زیرا important به عنوان اولین متغیر ، مقداردهی شده در حالی که متغیر somevar به عنوان اولین متغیر آدرس دهی شده ، باید مقداردهی شود ، در نتیجه هنگامی که شما این متغیر را که قبل از important

تعریف شده است مقداردهی می کنید ، مقدار somevar می تواند overwrite شود.این مشکل می تواند با اضافه کردن دو خط از خطر Buffer Overflow در امان بماند:

```
strcpy(somevar, argv[1]);
strcpy(important, "command");
```

این قسمتی از یک آسیب پذیری Buffer Overflow بود که در یک برنامه توضیح داده شد.شاید پیدا کردن یک آسیب پذیری به این شکل کار بسیار ساده ای باشد ، اما در نرم افزارهایی که شما هیچ اطلاعات ضروری در مورد آن ها برای یافتن آسیب پذیری ندارید ، کار بسیار دشواری خواهد بود.زبان C زبانی است که بر مبنای توابع امن نوشته نشده است و برخی توابع مهم آن نیز باعث به وجود آمدن چنین آسیب پذیری هایی می شوند.نام برخی از توابع در زیر آورده شده است:

1) Strcpy()
2) Strncat()
3) Strcat()
4) Strncat()
5) (v)sprint()
6) (v)sprintf()
7) Sscanf()
8) Vscanf()
9) Fscanf()
10) Gets()
11) Fgets()

قبل از استفاده از این توابع ، حتما باید مقدار و اندازه ی ورودی چک کرد و سپس از این توابع استفاده نمود.

در این مقاله از دو روش پیدا کردن آسیب پذیری استفاده شد:

1) پیدا کردن توابعی که بیشتر آسیب پذیری ها از آن ها ناشی می شوند یعنی همان توابع گفته شده در بالا
2) وارد کردن مقادیر طولانی به برنامه

در پیدا کردن آسیب پذیری در برنامه های مختلف ، ابتدا از این دو روش استفاده کنید و سپس به دنبال روش های دیگر بروید.

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