

سرریز بافر (Buffer Overflow) چیست ؟



نویسنده بهرنگ فولادی
Behrang@hat-squad.com

1382/8/18

www.hat-squad.com
گروه امنیتی Hat-Squad
(مشاور و ارائه دهنده سرویسهای امنیتی)
Service@hat-squad.com

Copyright © 2003, Hat-Squad Security Group

" درج تمام یا قسمتی از مطالب این مقاله تنها با اجازه نویسنده آن مقدور می باشد "

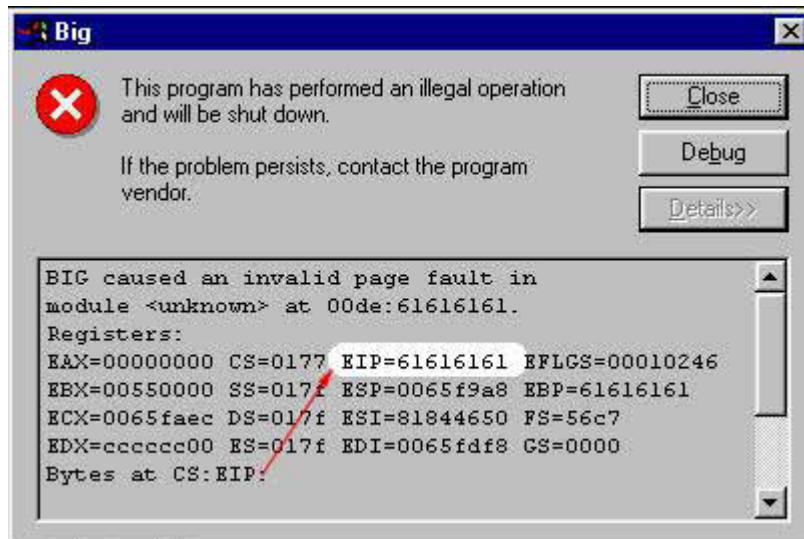
سرریز بافر (Overflow Buffer) از قدیمیترین مشکلات امنیتی سیستمهای کامپیوتری بوده است. در حال حاضر اگر به ضعفهای امنیتی نرم افزارهای مختلف که در سایتهایی مثل [SecurityFocus](#) ثبت شده اند، نگاهی بیاندازید، متوجه می شوید که حداقل 1/3 از این ضعفها مربوط به Buffer Overflow می شوند. این مشکل در تمام سیستم های عامل دیده شده است. در این مقاله با یک مثال ساده، موضوع Buffer Overflow و چگونگی استفاده نفوذگرها از آن جهت نفوذ به سیستم های کامپیوتری بررسی خواهد شد. در پایان روشهایی برای جلوگیری از این نوع حملات ارائه خواهد شد. برنامه های ذکر شده در این مقاله به زبان C نوشته شده اند و باید تحت سیستم عامل Windows کامپایل شوند(در این مقاله تنها سرریز بافر در برنامه های Win32 مورد نظر است و به سیستم های عامل دیگر مثل Unix/Linux اشاره نخواهد شد).

بررسی موضوع را با یک برنامه کوتاه آغاز می کنیم:

```
-----  
/* big.exe */  
#include <stdio.h>  
  
int insecure_func (char *big) {  
  
char insecure_buff[100];  
strcpy(insecure_buff, big);  
return 0;  
}  
  
int main (int argc, char *argv[]) {  
char input_buff[1024];  
gets(input_buff);  
insecure_func(input_buff);  
return 0;  
}  
-----
```

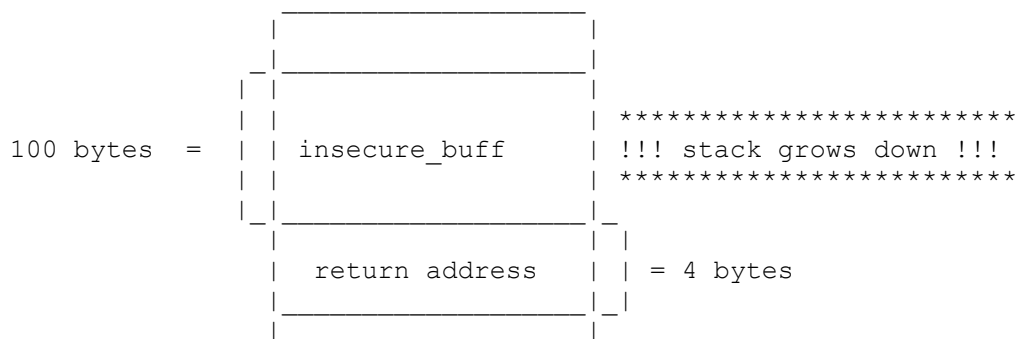
برنامه ابتدا رشته ورودی از صفحه کلید را در آرایه input_buff قرار میدهد سپس هنگامی که تابع **insecure_func** با فرستادن input_buff فراخوانی شود، این تابع مقدار موجود در input_buff را در **insecure_buff** کپی خواهد کرد. نکته اصلی کوچکتر بودن اندازه **insecure_buff** از **input_buff** است. بطوریکه اگر input_buff بیشتر از 100 کاراکتر را در خود جای داده باشد، insecure_buff سرریز (Overflow) خواهد شد.

حال برنامه را کامپایل و اجرا کنید و بیش از 100 کاراکتر 'a' را به عنوان ورودی به برنامه بدهید. پس از زدن کلید Enter پیغام خطایی شبیه مورد زیر دریافت خواهید کرد:



پیغام خطا

ببینیم چه اتفاقی افتاده است: هر گاه یک تابع از درون یک روال دیگر فراخوانی میشود، سیستم عامل آدرس برگشت به روال فعلی را در محلی از حافظه به نام "پشته" (Stack) قرار داده و کنترل را به روال فراخوانی شده می دهد. بدین ترتیب پس از پایان روال مذکور، سیستم عامل با بازیابی آدرس برگشت از Stack دوباره اجرای برنامه راه به روال اصلی می دهد. در مثال بالا آدرس برگشت قبل از وقوع سرریز (پیش از اجرای دستور strcpy) به دستور return 0 اشاره دارد. بنابراین قبل از وقوع سرریز Stack دارای ساختار زیر است:



ساختار Stack

پشته (stack) همیشه به سمت آدرسهای پایین تر رشد می کند. هنگام شروع اجرای برنامه، سیستم عامل 100 بایت برای بافر insecure_buff کنار می گذارد. واضح است که اجرا شدن دستور strcpy در حالتی که اندازه رشته موجود در big بزرگتر از insecure_buff باشد، باعث تغییر آدرس برگشت (return address) خواهد شد. اگر کمی دقت کنید، می فهمید که چرا آدرس برگشت بعد از سرریز x616161610 است. x061 کد اسکریپت کاراکتر 'a' در مبنای 16 است که توسط دستور strcpy قرار بوده به بافر insecure_buff کپی شود.

تا اینجا علت و چگونگی بروز overflow stack مشخص شد. اما چگونه این مشکل برای نفوذ به سیستم مورد استفاده قرار می گیرد؟ می دانیم که تمام برنامه ها و روالهایی که روی یک سیستم عامل در حال اجرا هستند، در آخرین لایه، چیزی جز کدهای ماشین نیستند که پشت سرهم خوانده و اجرا می شوند. CPU کامپیوتر آدرس حافظه مربوط به دستورالعمل بعدی را در طول اجرای برنامه از رجیستر EIP خوانده و کنترل اجرای برنامه را به آن آدرس منتقل می کند. حال اگر بتوان آدرس موجود در این رجیستر را در هر مرحله ای از اجرای برنامه به مقدار دیگری تغییر داد،

CPU بدون درنگ اجرای بقیه برنامه را از این آدرس جدید ادامه خواهد داد. تصور کنید در محل آدرس جدید کد يك backdoor یا سرویس پنهانی ویا هر نوع کد مخرب دیگری قرار داشته باشد. نتیجه این خواهد شد که کامپیوتر این کد را بجای کد برنامه اصلی که مسیرش توسط ما عوض شده، اجرا خواهد نمود و بدین ترتیب نفوذگر خواهد توانست با استفاده از backdoor یا کد مخرب اجرا شده، کنترل سیستم مزبور را بدست گیرد. بنابراین نفوذگر برای رسیدن به هدف خود باید دو مساله را حل کند. **اول یافتن راهی برای ایجاد overflow در سیستم هدف** . برای اینکار نفوذگر ، سرویسها و برنامه های در حال اجرا روی سیستم هدف مانند Server Web ، Mail Server ، Ftp Server و ... را برای یافتن روشی جهت overflow کردن هر کدام از آنها آزمایش خواهد نمود. بحث یافتن overflow ها از حوصله این مقاله خارج است و نمی توان روش استاندارد را پیشنهاد کرد و بیشتر روی سعی و خطا استوار است. **مرحله دوم استفاده از برنامه overflow شده برای اجرای کد دلخواه.**

برای نشان دادن روش کار نفوذگر، سعی می کنیم برنامه با استفاده از برنامه big.exe کد مورد نظر خود را روی سیستم اجرا کنیم. رشته ای را به عنوان ورودی (به جای 100 کاراکتر a) می سازیم که شامل يك کد کوچک دستورات زبان اسمبلی (که اصطلاحاً exploit نامیده می شود) است که کار مورد نظر ما را روی سیستم انجام خواهد داد و با تغییر آدرس برگشت، کنترل را به این کد خودمان می دهیم. بدین ترتیب به نتیجه مورد نظر خواهیم رسید.

برای نوشتن کد Exploit احتیاج داریم تا بدانیم رشته ورودی ساخته شده توسط ما در چه محلی روی آدرس برگشت ذخیره شده در انتهای Stack خواهد افتاد. برای دانستن این موضوع دو راه وجود دارد. روش اول استفاده از يك Disassembler برای یافتن اندازه بافر سرریز شده است. در این روش مجبوریم به دنبال تابعی بگردیم که سرریز در آن اتفاق می افتد. راه دوم انجام آزمون و خطا است ، ابتدا باید رشته کاراکتری از کد های اسکپی 32 تا 255 بسازیم ، کد کوچک زیر این کار را برآیمان انجام می دهد:

```
/* ascii.exe */
```

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

```
void main(void) {
```

```
    int i;
```

```
    for (i=0;i<256;i++) printf("%c",i);
```

```
}
```

این سوال پیش می آید که چرا کاراکترها با کد اسکپی بزرگتر از 32 را انتخاب کردیم. توجه داشته باشید که رشته ای که به عنوان ورودی به برنامه هدف می دهیم نباید حاوی کد های اسکپی کاراکترهای (CR(0x0c, (EOF(0x1a, (NULL(0x00 ، LF(0x0a باشد چون اگر تابع strcpy هنگام کپی کردن رشته کاراکتری به یکی از این کدها برسد آن را به عنوان انتهای رشته تلقی خواهد کرد و بقیه رشته کپی نخواهد شد. به همین دلیل است که ما بازه 32 تا 255 را که شامل هیچکدام از این کدها نیست انتخاب می کنیم. حال برنامه را کامپایل کرده ، به صورت زیر اجرا می کنیم :

```
C:\> c:\bof\big.exe | ascii.exe
```

این بار در پیغام خطا رجیستر EIP حاوی آدرس x8b8a89880 است (ترتیب قرارگیری از راست به چپ است). یعنی با شروع از محل **صدو چهارم** **بافر (x88-0x20=1040)** رشته ورودی روی آدرس برگشت می افتد. پس در رشته ای که خواهیم ساخت محل های 104 تا 107 حافظه (به طول 4 بایت) باید حاوی آدرس برگشت به کدی باشد که می خواهیم اجرا شود.

مشکل اول حل شد ، حالا باید تصمیم بگیریم که چگونه کد Exploit را تشکیل دهیم . در این باره دو امکان وجود دارد :

1- قرارگیری کد Exploit از ابتدای بافر تا محل 104 بافر

2- قرارگیری کد Exploit از محل 108 بافر به بعد

انتخاب روش اول اندازه کد Exploit مارابه 104 بایت محدود خواهد کرد ، به همین جهت روش دوم را انتخاب می کنیم و محل های حافظه قبل از آدرس صدوچهارم را نیز با کد دستور اسمبلی (NOP (No Operand یعنی x900 پر می کنیم.

مساله آخر تعیین آدرس محل حافظه است که می خواهیم به جای آدرس برگشت واقعی فرار دهیم. ابتدا به بررسی وضعیت رجیسترها و ساختار بافر درست قبل از اجرای دستور RET اسمبلی (تولید شده توسط دستور return 0) می پردازیم:



همان طور که می بینید ، درست قبل از اجرای دستور RET رجیستر ESP به محل 104 حافظه اشاره خواهد کرد ، پس اگر بتوانیم در این لحظه یک دستور jmp esp اجرا کنیم ، پردازنده بلافاصله کنترل را به آدرس 4 بایتی قرار داده شده در محل های 104 تا 107 می دهد و ما به منظور خود رسیده ایم . ابتدا باید حافظه سیستم را به دنبال کد دستور jmp esp (0xff0xe4) جستجو کنیم ، آدرس حافظه پیدا شده همان آدرسی است که بجای آدرس برگشت واقعی در محل های 104 تا 107 بافر قرار می گیرد. با این حساب ترتیب اجرای برنامه به صورت زیر در خواهد آمد:

RET--> JMP ESP--> Our Exploit Code

ترکیب xff0xe40 را می توان هم در حافظه برنامه big.exe و هم در حافظه مربوط به DLL های متصل به آن جستجو کرد. بهترین راه جستجو در DLL های متصل به برنامه است (در اینجا یکی دو فصل مربوط به فرمت فایل های PE ویندوز را رد می کنیم و مطالعه آن را به عهده خواننده علاقه مند می گذاریم). فایل های DLL سیستم در ویندوز NT با شروع از آدرس Image Base در حافظه Load می شوند با کمک برنامه های PE Analyser می توان به آسانی این آدرس را پیدا کرد. در اینجا از برنامه LISTDLLS استفاده می کنیم که می توانید آن را از سایت [sysinternals](http://sysinternals.com) دریافت کنید :

```
C:\bof> listdlls big.exe
```

Base	Size	Version	Path
0x00400000	0x27000		C:\bof\big.exe
0x77f60000	0x5c000	4.00.1381.0130	D:\WINNT\System32\ntdll.dll
0x77f00000	0x5e000	4.00.1381.0133	D:\WINNT\system32\kernel32.dll

اطلاعات راجع به محل قرارگیری برنامه و DLL های آن در حافظه را می بینیم. می توانیم در حافظه یا داخل هر کدام از این 3 فایل نشان داده شده به دنبال کد دستور jmp esp بگردیم. جستجو در حافظه راحتتر است چون نیازی به محاسبه Offset ها بر خلاف داخل فایل نیست. از یک Debugger مثلا SoftICE استفاده می کنیم و هنگام وقوع سرریز که کنسول SoftICE ظاهر می شود، دستور زیر را اجرا می کنیم:

```
S 1000000 | ffffffff fee4
```

یعنی جستجوی حافظه از آدرس x010000000 (اولین آدرسی که بایت اول مخالف صفر دارد) تا آخرین Offset حافظه یعنی xFFFFFFF0. نتیجه زیر حاصل خواهد شد:

```
Pattern found at 0023:77f327e5 (77f327e5)
```

دستور jmp esp در محل آدرس x77f327e50 پیدا شد این آدرس حاوی هیچکدام از کدهای (CR(0x0c, EOF(0x1a, NULL(0x00), LF(0x0a) نیست و براحتی می توان از آن استفاده کرد.

توجه: آدرس فوق با توجه به ورژن Service Pack نصب شده روی سیستم عامل هدف مقادیر متفاوتی خواهد بود، به این سبب در مواردی که Remote Exploit می نویسیم به طریقی از شماره Service Pack نصب شده روی کامپیوتر هدف اطلاع پیدا کنیم و سپس آدرس درست برای آن Service Pack را استفاده کنیم. روشهای پیشرفته دیگری برای Exploit نویسی وجود دارند که این مشکل را حل می کنند.

در این مرحله تمام اطلاعات لازم را در اختیار داریم: 1- محل قرارگیری کد Exploit معلوم شده است 2- آدرس که باید بجای آدرس برگشت واقعی قرار داده شود، می دانیم. تنها کار باقیمانده نوشتن برنامه Exploit است که بر حسب Local یا Remote بودن هدف متفاوت خواهد بود. در این زمینه مطالب و کدهای فراوانی روی وب موجود هستند که با پیش زمینه فعلی به آسانی می توانید از آنها ایده بگیرید.

چگونه از حملات Buffer Overflow جلوگیری کنیم؟

حملات buffer overflow از ضعف حاصل از عدم تست اندازه داده ورودی استفاده می کنند. اگر برنامه نویس big.exe قبل از دستور strcpy(insecure_buff, big); اندازه ورودی big را چک می کرد یا بجای تابع strcpy از strncpy به صورت strncpy(insecure_buff, big, 100); استفاده می کرد، مشکل overflow بروز نمی کرد. **لذا مهمترین اصل در برنامه نویسی یک سرویس یا برنامه مقاوم در برابر overflow ها چک کردن اندازه تمام ورودیها به برنامه قبل از انجام هر کاری روی داده هاست.** برای برنامه نویسان محیط linux/unix کتابخانه هایی مانند StackGuard وجود دارند که برنامه نویس می تواند با لینک کردن این کتابخانه ها به نرم افزارش، جلوی بسیاری از overflow ها را بگیرد.

اما در اکثر موارد، به کد سورس برنامه یا سرویس نصب شده روی سیستم خود دسترسی نداریم و قادر به تشخیص ضعفهای احتمالی از طریق بررسی سورس برنامه نیستیم همچنین اینکار از توان افراد غیر متخصص خارج می باشد. چاره ای که در بعضی سیستم های عامل مانند Sun/OS و Linux اندیشیده شده است، ممانعت از اجرای کد در محیط پشته (Stack) است. همچنین این روش بصورت محدودی روی سیستمهای ویندوز پیاده شده است. اما بهترین و راحتترین روش برای عموم نصب تمام Patch ها و Fix های ارائه شده توسط تولید کننده نرم افزار است که در 99% مواقع موثر خواهد بود.

سوالات و نظرات خود را به آدرس behrang@hat-squad.com ارسال نمایید .

دریافت اطلاعات بیشتر در مورد PE :

http://msdn.microsoft.com/library/en-us/dnwbgen/html/msdn_peeringpe.asp?frame=true

دریافت برنامه LISTDLLS :

<http://www.sysinternals.com/>