進め！リバースエンジニアリングの道 NEXT

第1回 セキュリティ機能

文●愛甲健二

次のステップへ

前回までの記事「進め！ リバースエンジニアリングの道」では、主にアセンブラを中心にソフトウェア解析の基礎を学習してきましたが、今回からはより実践的な技術を身に付け、CTF（Capture the Flag）やマルウェア解析といった分野でも十分に戦えるようなスキルセットの獲得を目指しましょう。

アセンブラは、確かにリバースエンジニアリングの基礎であり、必ず習得する必要がある技術です。しかしアセンブラだけ覚えていても、それはただのプログラマーであり解析者とは呼べません。アセンブラの技術はあくまでも求められるスキルの1つでしかなく、セキュリティ業界で戦うリバースエンジニアを目指すならば、その他にもさまざまな分野のスキルセットが必要になります。本連載では、そのようなセキュリティエンジニアに求められる解析技術に焦点を当てながら進めていきたいと思います。というわけで、読者の皆様、改めてよろしくお願い致します。

```
$ sudo su
[sudo] password for user:（パスワードを入力）
# echo 0 > /proc/sys/kernel/randomize_va_space
# cp test01 /tmp; cd /tmp; chmod 4755 test01
# ls -l test01
-rwsr-xr-x 1 root root 7242 2011-06-20 05:29
test01
# exit
$ cp exploit.py /tmp; cd /tmp
$ ./test01 `python exploit.py`
0xbffff7a0 [ENTER]
Segmentation fault
$ ./test01 `python exploit.py "bffff7a0"`
0xbffff7a0 [ENTER]
# whoami
root
```

図1　テスト用ファイルの設定

```
$ ./test01 `python exploit.py bffff6f0`
0xbffff700
Illegal instruction
$ ./test01 `python exploit.py
"bffff700"`AAAABBBBCCCCDDDD
0xbffff6f0
Segmentation fault
$ ./test01 `python exploit.py
"bffff6f0"`AAAABBBBCCCCDDDD
0xbffff6f0
# whoami
root
```

図2　権限の奪取

権限奪取の仕組み

まずは、脆弱性を利用した権限奪取の仕組みを説明します。付録DVD-ROMに収録されているファイル群をUbuntu Linux（こちらもisoファイルをDVDに収録しています。HTMLメニューの解説などを参考に、実機や仮想マシンで使用してください）にコピーし、端末から図1のコマンドを順番に入力してください。

最初にrootになり、セキュリティ機能ASLR（説明は後述）を無効にします。そしてtest01を/tmp以下にコピーし、setuidをrootで有効にします。これでtest01は他のユーザーが実行した際もroot権限で動作します。「ls -l」と入力してユーザーとグループが共にrootになっていることを確認してください。もしrootになっていなければ、「chown root test01」と入力し、ユーザーをrootに変更してください。

この状態で一般ユーザーに戻り、 今度はexploit.pyを/tmp以下にコピーします。そしてtest01の引数にexploit.pyの出力結果を渡します。 その際、1度目の実行でtest01が出力したアドレスbffff7a0を、2度目の実行時にexploit.pyの引数として渡してください(図2)。もし出力されたアドレスの中に0x00が含まれていた場合は、引数に渡す文字列の長さを変えることで出力されるアドレスが変わるため、うまく0x00がない状態のアドレスを出力させてください。

以上でroot権限で/bin/shが実行され、一般ユーザーからrootを奪取できました。test01のソースコードはtest01.sであり、test.cを元に作られていますので、興味があればご参照ください。一般的なスタックバッファオーバーフローの脆弱性となっています。

セキュリティ機能

さまざまな攻撃手法が研究され始めると、それと同時に「いかにして守るか」といった防御手法も考案されます。10年前のLinuxと現在のLinuxでは、ディストリビューションにもよりますが、明らかに現在のLinuxの方がセキュアだと言えます。それは、多くのセキュリティエンジニアが攻撃手法を確立し、またそれらから守る手段を研究し続けたからこその結果です。そして、 その研究成果のいくつかはUbuntu Linuxにも取り入れられています。

第1回目の今回は、Ubuntu Linuxを守っている「OSのセキュリティ機能」について調べていきましょう。 今回取り扱うセキュリティ機能は「Address Space Layout Randomization」「Exec-Shield」そして「StackGuard」の3つです。どれも現在においては一般的なセキュリティ機構ですので、 今後のためにもぜひ仕組みを理解してください。

ASLR

ASLR(Address Space Layout Randomization)は、スタックや各モジュール、動的に確保したメモリ(ヒープ)などのアドレスをランダムに決定する仕組みです。これはOSの機能であり、Ubuntu Linuxでは/proc/sys/kernel/randomize_va_spaceで設定を確認、変更できます。rootになり、端末で以下のコマンドを実行してください。

```
# cd /proc/sys/kernel/
# cat randomize_va_space
2       ←有効:デフォルト
# echo 0 > randomize_va_space
# cat randomize_va_space
0       ←無効
```

catコマンドでrandomize_va_spaceの値を確認すると、0、1、2のいずれかの値が出力されます。簡単に説明すると、0は無効、1はヒープ以外をランダム化、2はすべてランダム化であり、Ubuntu Linuxではデフォルトで2に設定されています。

DVD-ROMに収録されているtest00は、スタックとmallocで確保したメモリのアドレスを表示するプログラムです。ASLRが無効だとtest00を何度実行してもアドレスが変わりませんが、ASLRを有効にしていると、1度目と2度目でアドレスが変わっています。これがASLRの具体的な動作です。

■ASLR無効

```
$ ./test00       ←1度目
malloc: 0x804b008
 stack: 0xbffff7d8
$ ./test00       ←2度目
malloc: 0x804b008
 stack: 0xbffff7d8
```

■ASLR有効

```
$ ./test00
malloc: 0x87f9008
 stack: 0xbfac7938
$ ./test00
malloc: 0x8bc4008
 stack: 0xbfdeeea8
```

さて、先ほどtest01を利用してroot権限を奪取した際、あらかじめASLRを無効にしておきました。ではASLRが有効だった場合は、ど

のようになるでしょうか。試してみましょう。

```
$ ./test01 `python exploit.py`
0xbfd07cc0
Segmentation fault
$ ./test01 `python exploit.py
"bfd07cc0"`
0xbfc77df0
Segmentation fault
$ ./test01 `python exploit.py
"bfc77df0"`
0xbfa94e00
Segmentation fault
```

ASLRが有効だった場合、test01が表示するアドレスが毎回ランダムになります。そのため正確なアドレス値をexploit.pyに渡すことができず、結果的に権限奪取に成功しません。そもそもなぜexploit.pyにアドレス値を渡さなければならないのか？ このアドレス値はいったい何か？ といった疑問はさておき、とりあえずは「ASLRが有効だと攻撃が成功しにくくなる」といった事実は確認できたと思います。

今回はASLRを紹介することが目的であるため、ここでひとまず説明を終えますが、ASLRの有用性は、連載が進み実際にExploitを記述、解説する際に改めて説明させていただきたいと思います。とりあえずは「なるほど、このような機能があるんだ、へー」という程度の認識で構いません。

Exec-Shield

Exec-Shieldとは、簡単に言えば「実行コードが置かれるメモリ空間以外には、極力実行権限を持たせない」という仕組みです。例えば、スタックとして使用されているメモリ空間に実行すべきマシン語が置かれることは通常あり得ませんので、スタック領域は読み書きのみを許可して実行を不可にする、逆にコードセクションにはマシン語が置かれていますが、そのマシン語を書き換える必要性は一般的なソフトウェアではまずないので、そのメモリ空間は書き込みを不可にする、といった具合に、置かれているデータによって、メモリの読み書き実行権限を設定します。

任意のプログラム内のメモリ領域の読み書き実行権限を確認するには、/proc/<PID>/mapsを出力します。

■端末1

```
$ ./test02
0xbffff800
```

■端末2

```
$ ps -aef | grep test02
kenji     16918 15209  0 05:05
pts/2     00:00:00 ./test02
kenji     16920 15291  0 05:06
pts/3
00:00:00 grep --color=auto test02
$ cat /proc/16918/maps | grep
stack
bffeb000-c0000000 rw-p 00000000
00:00 0           [stack]
```

test02は、test01にExec-Shield機能を追加したものです。スタックとしてbffeb000～c0000000のアドレスが使用されていますが、rw-pと実行権限を意味するxがありません。

では、ASLRを無効にし、Exec-Shieldの効果を試してみましょう。

```
$ ./test02 `python exploit.py
"bffff7a0"`
0xbffff7a0
Segmentation fault
```

入力と出力のアドレスが一致しているにも関わらず、攻撃が成功しませんでした。

test02のソースコードであるtest02.sを読むと、test02の引数に渡されるデータはstrcpyによりスタック領域にコピーされますが、test02の/proc/<PID>/mapsを見ると、そのスタック領域には実行権限がありません。

つまり、スタックに展開されたデータ（/bin/shを実行するマシン語）は、確かにtest02のメモリ空間には存在しますが、実行ができないため、アドレスが一致しているのも関わらず、Segmentation faultで強制終了したわけで

す。

ASLRがアドレスを一致させない防御手法だったのに対し、Exec-Shieldは一致しても任意のマシン語を実行させない防御手法と言えます。

ちなみにtest01の/proc/<PID>/mapsを見ると、スタック領域にも実行権限が付いています。これがtest01とtest02の違いです。より詳細が知りたい方は、test01.sとtest02.sを見比べてみてください。

StackGuard

StackGuardはコンパイラの機能であり、コンパイル時に各関数の入口と出口にスタックが破壊されたことを検知するマシン語を挿入します。このような仕組み上、StackGuardは、ASLRやExec-Shieldといったセキュリティ機能とは異なり、異常をプログラム内で「検知」できます。

では、実際に試してみましょう。ASLRを無効にして、test03を実行します。

```
$ ./test03 `python exploit.py
"bffff7a0"`
0xbffff7a0
*** stack smashing detected ***:
./test03 terminated
======= Backtrace: =========
./test03[0x8048515]
[0xbffff7a0]
======= Memory map: ========
Aborted
-----
```

ASLRやExec-Shieldがの機能（Segmentation fault）によって止められていたのに対し、StackGuardはtest03自身が異常を検知し、Abortしています。

test03にはスタックバッファオーバーフローの脆弱性があり、 それによりスタック内でデータあふれが発生した際、StackGuardのコードがそのデータあふれを検知して、stack smashing detectedと表示し、プログラムを強制終了したわけです。test03.sを読むと、StackGuardのコードが確認できます。

■ 関数の入口

```
movl  %gs:20, %eax
movl  %eax, 60(%esp)
```

■ 関数の出口

```
movl  60(%esp), %edx
xorl  %gs:20, %edx
je    .L5
call  __stack_chk_fail
.L5:
```

%gs:20にはプログラム実行ごとにランダムな値が入っており、 それをスタックの終端60(%esp)にコピーします。60(%esp)以降にはebpやretがあるため、それを守るように配置されるわけです。そして、関数を出る前に、%gs:20と60(%esp)を比較します。もしスタックが何かしらの原因でオーバーフローしていれば、ebpやretと同じく、60(%esp)も書き換わっているはずなので、 それを検知し、__stack_chk_failへジャンプして処理を終了させます。

要するに、StackGuardはebpやretを守るための仕組みであり、典型的なスタックバッファオーバーフローに対しての防御手法と言えます。

C言語のコードをコンパイルする際は、gccはデフォルトでStackGuardのコードを付加しますが、-fno-stack-protectorオプションを付けることでStackGuardを無効にできます。

これらの機能を迂回するには？

今回3つのセキュリティ機能を紹介しましたが、これらは多くのOS、コンパイラで当たり前に実装されています。ソフトウェアはプログラマーの知らないところで守られ、バグがそのまま「任意のコード実行」につながらないよう多くの工夫がされています。 例えば、test04はStackGuardとExec-Shieldを有効にしたもので、これに加えてASLRをOS側で有効にすれば、仮にtest.cのようなプログラムを書いてしまったとしても、任意のコードを実行されることは難しくなります。

次回は、 これらの技術を踏まえた上でCTFの問題に挑戦していくことにしましょう。