Binary Reverse Engineering And Analysis

Course 8: Heap Exploitation on Linux

Caragea Radu

February 12, 2021

Recap

- Stack Smashing Protection
- Dynamic linking, the GOT and RELRO
- Write-What-Where conditions

Today

- What can still be exploited?
- A brief example of modern (2018-2019) vulnerabilities
- Highlight the exploitation method

Heap and allocators

- Compilers do a good job protecting the stack
- However, the heap is less hardened

Heap and allocators

- Compilers do a good job protecting the stack
- However, the heap is less hardened
- Many types of allocators exist:
  - Glibc: ptmalloc2 with or without tcache
  - Android: dlmalloc / jemalloc
  - Windows: segment heap or NT heap

Heap and allocators

- Compilers do a good job protecting the stack
- However, the heap is less hardened
- Many types of allocators exist:
  - Glibc: ptmalloc2 with or without tcache
  - Android: dlmalloc / jemalloc
  - Windows: segment heap or NT heap
- Code is always being added
- Sometimes, without thinking about security

Case study: Ubuntu 18.04

- New (rushed) features in the allocator
- Horrendous bugs in everyone's systems
- Let's investigate just one

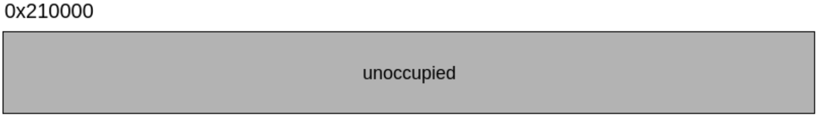
Malloc/Free

- Any dynamic memory allocation will ultimately use malloc
- Malloc, in turn, uses the heap segment. How?

Malloc/Free

- Any dynamic memory allocation will ultimately use malloc
- Malloc, in turn, uses the heap segment. How?
- Free: keeps lists of chunks for later reuse (by size)
- Malloc: retrieves an older chunk or creates a new one

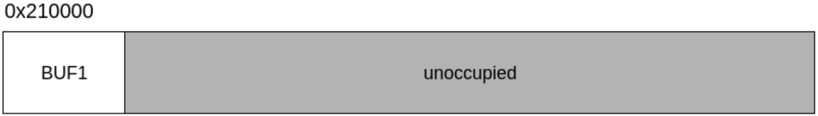
Initial state of Heap segment



0x210000

unoccupied

Allocations



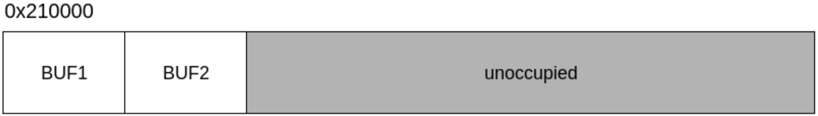
0x210000

BUF1

unoccupied

```
char *buf1 = malloc(0xf0); //0x210010
```

Allocations



0x210000

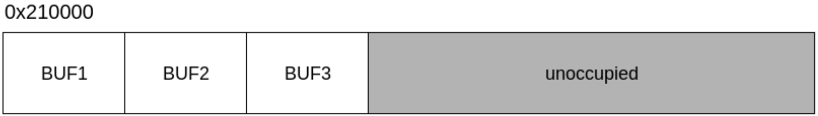
BUF1

BUF2

unoccupied

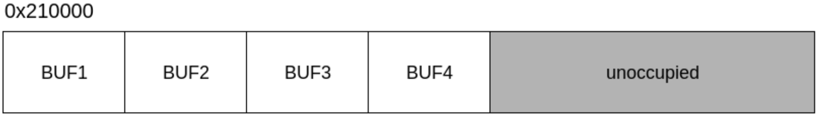
```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
```

Allocations



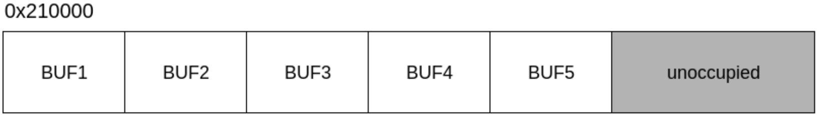
```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
```

Allocations



```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
char *buf4 = malloc(0xf0); //0x210310
```

Allocations

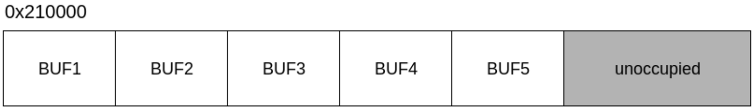


0x210000

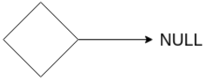
BUF1 | BUF2 | BUF3 | BUF4 | BUF5 | unoccupied

```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
char *buf4 = malloc(0xf0); //0x210310
char *buf5 = malloc(0xf0); //0x210410
```

Free and the free list

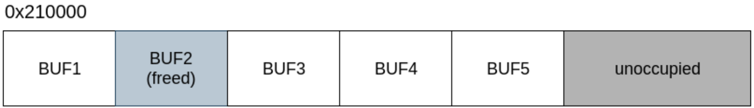


```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
char *buf4 = malloc(0xf0); //0x210310
char *buf5 = malloc(0xf0); //0x210410
```

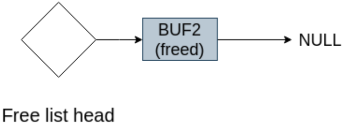


Free list head

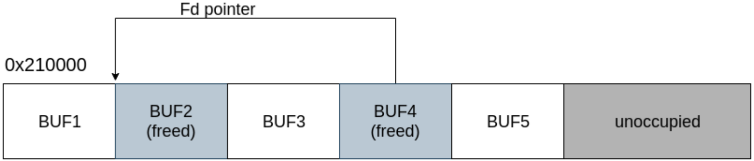
Free and the free list



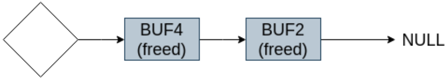
```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
char *buf4 = malloc(0xf0); //0x210310
char *buf5 = malloc(0xf0); //0x210410
free(buf2); //add to free list, link to NULL
```



Free and the free list



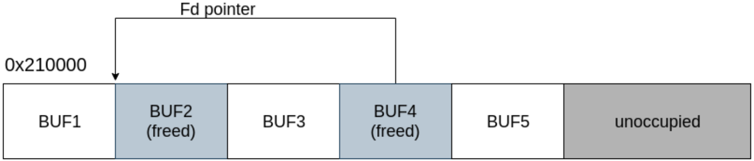
```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
char *buf4 = malloc(0xf0); //0x210310
char *buf5 = malloc(0xf0); //0x210410
free(buf2); //add to free list, link to NULL
free(buf4); //add to free list, link to buf2
```



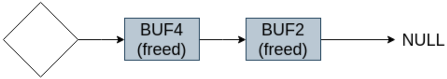
Possible attacks

- In large code bases, bugs inevitably surface
- Static code analyzers cannot always discover misuses
- Crashes are sometimes found but dismissed as unexploitable
- Let's see what happens when a pointer gets freed two times by accident

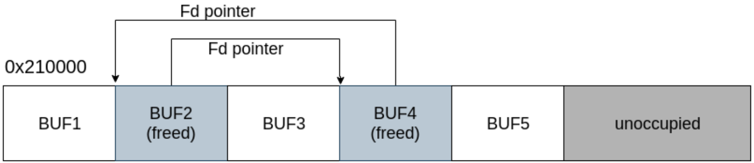
Double free attack (before corruption)



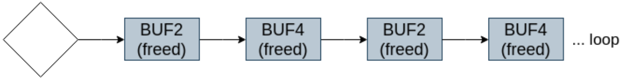
```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
char *buf4 = malloc(0xf0); //0x210310
char *buf5 = malloc(0xf0); //0x210410
free(buf2); //add to free list, link to NULL
free(buf4); //add to free list, link to buf2
```



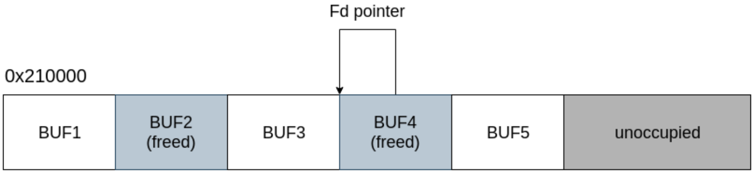
Double free attack (var 1)



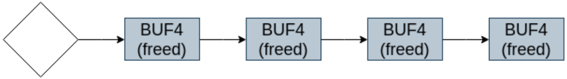
```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
char *buf4 = malloc(0xf0); //0x210310
char *buf5 = malloc(0xf0); //0x210410
free(buf2); //add to free list, link to NULL
free(buf4); //add to free list, link to buf2
free(buf2); //add to free list, link to buf4
```



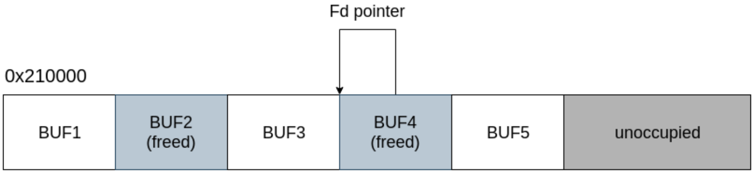
Double free attack (var 2)



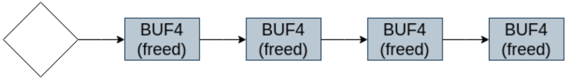
```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
char *buf4 = malloc(0xf0); //0x210310
char *buf5 = malloc(0xf0); //0x210410
free(buf2); //add to free list, link to NULL
free(buf4); //add to free list, link to buf2
free(buf4); //add to free list, link to buf4
```



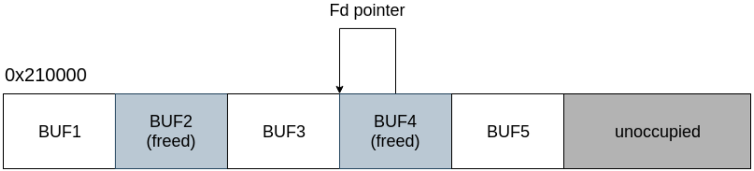
Double free attack (var 2)



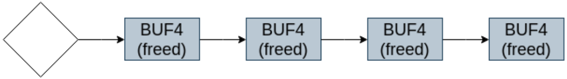
```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
char *buf4 = malloc(0xf0); //0x210310
char *buf5 = malloc(0xf0); //0x210410
free(buf2); //add to free list, link to NULL
free(buf4); //add to free list, link to buf2
free(buf4); //add to free list, link to buf4
char *buf6 = malloc(0xf0); //0x210310 (buf4)
```



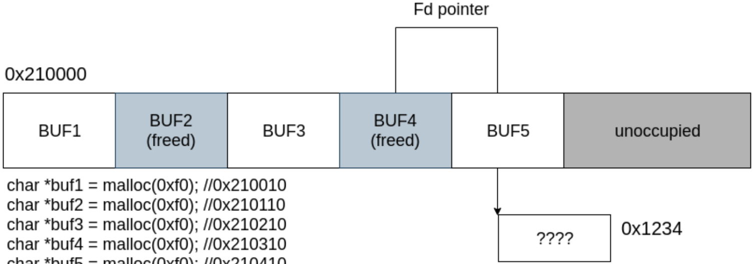
Double free attack (var 2)



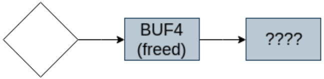
```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
char *buf4 = malloc(0xf0); //0x210310
char *buf5 = malloc(0xf0); //0x210410
free(buf2); //add to free list, link to NULL
free(buf4); //add to free list, link to buf2
free(buf4); //add to free list, link to buf4
char *buf6 = malloc(0xf0); //0x210310 (buf4)
char *buf7 = malloc(0xf0); //0x210310 (buf4)
```



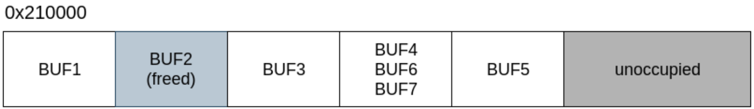
Double free attack (var 2)



```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
char *buf4 = malloc(0xf0); //0x210310
char *buf5 = malloc(0xf0); //0x210410
free(buf2); //add to free list, link to NULL
free(buf4); //add to free list, link to buf2
free(buf4); //add to free list, link to buf4
char *buf6 = malloc(0xf0); //0x210310 (buf4)
*(int*)buf6 = 0x1234; // fd pointer = 0x1234
```



Double free attack (var 2)



0x210000

BUF1

BUF2
(freed)

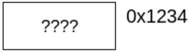
BUF3

BUF4
BUF6
BUF7

BUF5

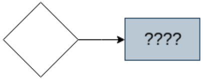
unoccupied

```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
char *buf4 = malloc(0xf0); //0x210310
char *buf5 = malloc(0xf0); //0x210410
free(buf2); //add to free list, link to NULL
free(buf4); //add to free list, link to buf2
free(buf4); //add to free list, link to buf4
char *buf6 = malloc(0xf0); //0x210310 (buf4)
*(int*)buf6 = 0x1234; // fd pointer = 0x1234
char *buf7 = malloc(0xf0); //0x210310 (buf4)
```



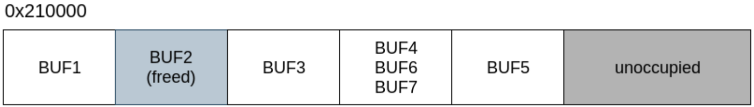
????

0x1234

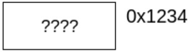


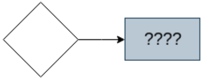
????

Double free attack (var 2)



```
char *buf1 = malloc(0xf0); //0x210010
char *buf2 = malloc(0xf0); //0x210110
char *buf3 = malloc(0xf0); //0x210210
char *buf4 = malloc(0xf0); //0x210310
char *buf5 = malloc(0xf0); //0x210410
free(buf2); //add to free list, link to NULL
free(buf4); //add to free list, link to buf2
free(buf4); //add to free list, link to buf4
char *buf6 = malloc(0xf0); //0x210310 (buf4)
*(int*)buf6 = 0x1234; // fd pointer = 0x1234
char *buf7 = malloc(0xf0); //0x210310 (buf4)
char *buf8 = malloc(0xf0); //0x1234 !!!
```





Write What Where

- `https://cwe.mitre.org/data/definitions/123.html`
- Hello, old friend!

Aside

- Bugs depend on allocator implementation and checks
- The more checks, the slower the allocator
- Before Ubuntu 18.04

Aside

- Bugs depend on allocator implementation and checks
- The more checks, the slower the allocator
- Before Ubuntu 18.04 this bug is exploitable but some asserts must be passed
- After Ubuntu 18.04

Aside

- Bugs depend on allocator implementation and checks
- The more checks, the slower the allocator
- Before Ubuntu 18.04 this bug is exploitable but some asserts must be passed
- After Ubuntu 18.04 this bug is still exploitable but some asserts must be passed

Other heap vulnerabilities

- Buffer used after free: similar metadata corruption possible
- Buffer not initialized properly: data can ”resurface” (info leak)
- Many other allocator-specific vulnerabilities

Practice

- Any Questions?
- `http://pwnthybytes.ro/unibuc_re/08-lab.html`