Lec01: Stack Overflow and Protections

Taesoo Kim

Goals and Lessons

- Learn about the stack overflow bugs
- Understand their security implications (i.e., control hijacking)
- Understand the off-the-shelf mitigation (i.e., ssp, DEP, RELO, ASLR)
- Learn them from the real-world examples (i.e., qemu/ruby/wireshark)

“Smashing The Stack For Fun And Profit”

CS101: What's Wrong?

```
main() {
  char buf[16];
  scanf("%s", buf);
}
```

CS101: What's Wrong?

```
main() {
  char buf[16];
  scanf("%s", buf);
}
```

1. scanf("%s", &buf)
2. scanf("%16s", buf)
3. scanf("%15s", buf)
4. scanf("%as", &bufptr)

Error-prone C APIs: scanf()

- scanf("%15s", buf) // CORRECT!

```
$ cd lec01-stackovfl/apis
$ cat scanf.c
$ make
$ ./scanf
...
```

Error-prone C APIs: scanf()

- scanf("%16s", buf) // BUG!

```
$ ./scanf
aaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaa
01: 61 (a)
02: 61 (a)
...
0e: 61 (a)
0f: 61 (a)
10: 61 (a)
11: 00 ()
12: BE ()
13: AD ()
14: DE ()
```

Control Hijacking Attack: Basic Idea

```
main() {
  char buf[16];
  scanf("%s", buf); // BUG!
}
```

```
(top, growing)
  |<--- current frame -->|<-- caller-->|
  [buf             ][fp][ra][...         ]
  |<----  16 --->|
```

Control Hijacking Attack: Basic Idea

```
main() {
  char buf[16];
  scanf("%s", buf); // BUG!
}

(top, growing)
  |<--- current frame -->|<-- caller-->|
  [buf           ][fp][ra][...         ]
  |<----  16 --->|
  AAAAAAAAAAAAAAAABBBBCCCC... =>

  !SEGFAULT @eip=CCCC
```

Control Hijacking Attack: Where to Jump?

- Jump to the injected code (e.g., stack, environ, etc)

```
(top, growing)
  |<--- current frame -->|<-- caller-->|
  [buf          ][fp][ra][...         ]
  |<----  16 --->|
  AAAAAAAAAAAAAAAABBBBXXXX[shellcode ..]
                    |    ^
                    +----+ (1) attacker's input
```

Control Hijacking Attack: Where to Jump?

- Jump to the injected code (e.g., stack, environ, etc)

```
(top, growing)
  |<--- current frame -->|
  [buf           ][fp][ra] ... [SHELLCODE=...]
  |<----  16 --->|              ^
  AAAAAAAAAAAAAAAABBBBXXXX      |
                    |           |
                    +---------+ (2) crafted environ
```

Control Hijacking Attack: Advanced Topics

- Stack pivoting when frame pointer is crafted
- Even off-by-one (e.g., writing NULL) is enough for stack pivoting

```
(top, growing)
  |<--- current frame -->|<-- caller-->|
  [buf           ][fp][ra][...        ]
  |<----  16 --->|
(1)AAAAAAAAAAAAAAAAXXXX -> enough to control (i.e., leave; ret)
(2)AAAAAAAAAAAAAAAAX    -> off-by-one
```

Memory Safety in C/C++

- Spatial safety → e.g., buffer over/underflow
- Temporal safety → e.g., use-after-free

Addressing Memory Safety Issues in C/C++

- Spatial safety → e.g., buffer over/underflow
  - → Tracking object boundaries and verifying all memory accesses
- Temporal safety → e.g., use-after-free
  - → Tracking life time of objects and verifying all memory accesses

Idea in C/C++: if we implement everything *correctly*, we have opportunities to make the program much efficient (in terms of memory usage) and faster (in terms of execution speed)!

Error-prone C APIs: strncpy()

- strncpy(char *dest, const char *src, size_t n)

```
// BUG!
char buf[BUFSIZ];
strncpy(buf, input, sizeof(buf));
```

Error-prone C APIs: strncpy()

- strncpy(char *dest, const char *src, size_t n)

```
char buf[BUFSIZ];
strncpy(buf, input, sizeof(buf) - 1);
buf[sizeof(buf) - 1] = '\0';
```

Error-prone C APIs: strncpy()

- NULL on the remaining bytes (why?)
- strncpy(buf, “A”*len(buf), buf) → leaving buf non-NULL-terminated
- dest and src should not be overlapping
- Return dest, not #chars copied!

```
char* strncpy(char *dest, const char *src, size_t n) {
  size_t i;
  for (i = 0; i < n && src[i] != '\0'; i++)
    dest[i] = src[i];
  for ( ; i < n; i++)
    dest[i] = '\0';
  return dest;
}
```

Error-prone C APIs: strncat()

- dest is always NULL-terminated C-string
- Copy max n + 1 (w/ null)! → strncat(dest, src, len - 1)
- Return dest, not #chars copied!

```
char* strncat(char *dest, const char *src, size_t n) {
  size_t dest_len = strlen(dest);
  for (size_t i = 0 ; i < n && src[i] != '\0' ; i++)
    dest[dest_len + i] = src[i];
  dest[dest_len + i] = '\0';
  return dest;
}
```

Suggestion for C-string manipulation

- Use: snprintf(buf, sizeof(buf), …)
- Alternatives: strlcpy(), strlcat()
    - Return #chars copied, or strlen(dest)
    - NULL-terminated, unless dest is full in cast of strlcat()

Outline

- Real-world examples:
    1. CVE-2017-15118: QEMU
    2. CVE-2014-4975: Wireshark
    3. CVE-2015-7547: glibc getaddrinfo()*
- Off-the-shelf defenses:
    1. Stack shield/canary (a.k.a, SSP in gcc)
    2. DEP (NX, W^X)
- Advance defenses: Shadow stack and CET

CVE-2017-15118: QEMU

```c
#define NBD_MAX_NAME_SIZE 256

static int nbd_negotiate_handle_info(...) {
  char name[NBD_MAX_NAME_SIZE + 1];
  uint32_t namelen;

  nbd_read(client->ioc, &namelen, sizeof(namelen), errp);
  nbd_read(client->ioc, name, namelen, errp);
}
```

CVE-2014-4975: Ruby

```c
void encodes(..) {
    char buff[4096];
    while (len >= 3) {
        while (len >= 3 && sizeof(buff)-i >= 4) {
            buff[i++] = ..; /* 4 times */;
            s += 3;
            len -= 3;
        }
        if (sizeof(buff)-i < 4) { /* flush */ }
    }
    if (len == 2) { ... }
    else if (len == 1) { ... }

    if (tail_lf) buff[i++] = '\n';
    /* flush */
}
```

Then, How to Prevent Stack Overflow?

- Two approaches:
  - Bug prevention (i.e., correct bound checking)
  - Exploitation mitigation (i.e., making exploit harder)

1. Prevent the buffer overflow at the first place
   - (e.g., code analysis, designing better APIs)
2. Protect "integrity" of ra, funcptr, etc (code pointers)
   - (e.g., exploitation mitigation → NX, canary)

Defense 1: Stack Canary

Stack Canary: Basic Idea

- Use a canary value as an indicator of the integrity of fp/ra
- Check the canary value right before using fp/ra (i.e., ret)

```
(top, growing)
  |<--- current frame -->|<-- caller-->|
  [buf           ][canary][fp][ra][...         ]
  |<----  16 --->|
  AAAAAAAAAAAAAAAA XOXOXO BBBBCCCC... =>
                   (corrupted?)
```

Subtle Design Choices for the Stack Canary

1. Where to put? (e.g., right above ra? fp? local vars?)
2. Which value should I use? (e.g., secret? random? per exec? per func?)
3. How to check its integrity? (e.g., xor? cmp?)
4. What to do after you find corrupted? (e.g., crash? report?)

GCC's Stack Smashing Protector (SSP)

- Options: -fstack-protector/all/strong/explicit
  - Scope: all >> strong >> default >> explicit
  - e.g., use of alloca(), buffer, array, etc

```
$ cd lec01-stackovfl/ssp
$ cat ssp.c
$ make check-sspopts
...
```

Case Study: Using SSP in Linux (> 3.14)

- -fstack-protector:
  - 0.33% larger code size
  - 2.81% of the functions covered
- -fstack-protector-strong:
  - 2.4% larger code size
  - 20.5% of the functions covered

ref. https://lwn.net/Articles/584225/

SSP in Detail: Instrumentation

```
int func1_benign(int arg) { return arg; }

$ ./check-func.py ssp-explicit func1_benign
func1_benign()@ssp-explicit
  push   rbp
  mov    rbp,rsp
  mov    DWORD PTR [rbp-0x4],edi
  mov    eax,DWORD PTR [rbp-0x4]
  pop    rbp
  ret
```

SSP in Detail: Instrumentation

```
$ ./check-func.py ssp-all func1_benign
func1_benign()@ssp-all
  push   rbp
  mov    rbp,rsp
  sub    rsp,0x20
  mov    DWORD PTR [rbp-0x14],edi
! mov    rax,QWORD PTR fs:0x28     // read canary @TLS
! mov    QWORD PTR [rbp-0x8],rax   // put it right above fp
! xor    eax,eax                   // clear it off
  mov    eax,DWORD PTR [rbp-0x14]
! mov    rdx,QWORD PTR [rbp-0x8]   // fetch canary on stack
! xor    rdx,QWORD PTR fs:0x28     // compare it with @TLS
! je     func1_benign+0x31
! call   __stack_chk_fail@plt      // stack smashed!
  leave
  ret
```

SSP in Detail: In Action

- Any interesting byte in canary?

```
$ ./canary
0x7ffd4d8e4278: 0xa                 // saved arg
0x7ffd4d8e4280: 0x100000000         // dummy
0x7ffd4d8e4288: 0xdace23a197bb3d00  // canary
0x7ffd4d8e4290: 0x7ffd4d8e42c0      // fp
0x7ffd4d8e4298: 0x55d3082a51fc      // ra
```

About "Terminator" Canary

- Why is the terminator canary special?
  - 0x0d000aff: NULL (0x00), CR (0x0d), LF (0x0a) and EOF (0xff)
- SSP: Used to contain NULL/EOF/LF (06/2014, see commit)
- SSP: Only contains NULL (@MSB) in a recent version

SSP: __stack_chk_fail()

- Immediately abort the program like below:

```
$ cd lec01-stackovfl/ssp
$ ./ovfl
*** stack smashing detected ***: ./ovfl terminated
Aborted
```

SSP: __stack_chk_fail() Implementation

```c
// @debug/stack_chk_fail.c
void __attribute__ ((noreturn))
__stack_chk_fail (void) {
  __fortify_fail ("stack smashing detected");
}

void __attribute__ ((noreturn))
__fortify_fail (const char *msg) {
  __libc_message (2, "*** %s ***: %s terminated\n",
                  msg, __libc_argv[0] ?: "<unknown>");
}
```

SSP: Security Issue in __stack_chk_fail()

- CVE-2010-3192: Arbitrary read after stack smashing
  - __libc_argv[0] is under control
  - Breaking confidentiality, e.g., leaking private keys

SSP: New Implementation

```
      /* Don't pass down __libc_argv[0] if we aren't doing bac
         since __libc_argv[0] may point to the corrupted stack
      __libc_message (need_backtrace ?
                        (do_abort | do_backtrace) : do_abort,
              "*** %s ***: %s terminated\n",
              msg,
              (need_backtrace && __libc_argv[0] != NULL
               ? __libc_argv[0] : "<unknown>"));
```

SSP: Placing Local Variables

```
long var1[32] = {1, };
int (*var2)(const char *) = system;
long var3 = 3;
```

```
$ cd lec01-stackovfl/ssp
$ make check-loc
func1_benign():
  0x7ffd6375c580: 0x1 (var1)
  0x7ffd6375c588: 0x2 (var2)
  0x7ffd6375c590: 0x3 (var3)
...
func5_buf_and_funcptr():
  0x7ffd6375c480: 0x7f5a62ecf380 (var2)
  0x7ffd6375c488: 0x3 (var3)
  0x7ffd6375c490: 0x1 (var1)
```

Limitation of Canary-based Approaches

1. Unprotected local variables (e.g., index, func ptrs)
2. Incrementally overwriting one byte at a time (in remote, fork())
3. Leaked canary (per execution)

Defense 2: DEP (NX, W^X)

- Data Execution Prevention (DEP)
    - aka, Non eXecutable, Writable ^ eXecutable
    - Basically, don’t make writable region executable at the same time

```
cat /proc/self/maps
5606bdf09000-5606bdf0b000 r--p /usr/bin/cat
5606bdf0b000-5606bdf0f000 r-xp /usr/bin/cat
5606bdf13000-5606bdf14000 rw-p /usr/bin/cat
...
5606bef45000-5606bef66000 rw-p [heap]
7ffcd93c8000-7ffcd93ea000 rw-p [stack]
7ffcd93f7000-7ffcd93fa000 r--p [vvar]
7ffcd93fa000-7ffcd93fc000 r-xp [vdso]
```

Advance Defense 1: Shadow Stack

- Option: -fsanitize=shadow-call-stack
    - Replicate return addresses in a safe place, so called shadow stack
    - The shadow stack directly indicates modification of return addresses

```
       (top) +--------------+
             |              v XXXX
<stack>  : [buf][canary][fp][ra][var][canary][fp][ra] ... (@rsp)
<shadow> :                  [ra]                  [ra] ... (@gs)
```

Advance Defense 1: Shadow Stack

```
void vuln(char *arg) { char buf[32]; ... }
```

```
$ cd lec01-stackovfl/safestack
$ make check-shadowstack
-vuln()@shadowstack-no
+vuln()@shadowstack-yes
+   mov    r10,QWORD PTR [rsp]
+   xor    r11,r11
+   add    QWORD PTR gs:[r11],0x8
+   mov    r11,QWORD PTR gs:[r11]
+   mov    QWORD PTR gs:[r11],r10
    push   rbp
    mov    rbp,rsp
    sub    rsp,0x40
...
```

Advance Defense 2: Safe Stack

- Option: -fsanitize=safe-stack
- Two stacks: safe/unsafe stacks for sensitive/non-sensitive data
  - Preventing stack overflow to the sensitive data
  - Disentangling the leakage of stack pointers

```
          (top)
<stack>  : [buf][canary][fp][ra][var][canary][fp][ra] ... (@rsp)

<safe>   : [fp][ra][fp][ra] ...                            (@rsp)
<unsafe> : [buf][canary][var][canary] ...                  (@fs)
            ======> overflow doesn't affect fp/ra + other sensative data
```

Advance Defense 2: Safe Stack

```
void vuln(char *arg) { char buf[32]; ... }

$ cd lec01-stackovfl/safestack
$ make check-safestack
-vuln()@safestack-no
+vuln()@safestack-yes
   push   rbp
   mov    rbp,rsp
-  sub    rsp,0x40
+  sub    rsp,0x20
+  mov    rax,QWORD PTR [rip+0x8271] ; read the base of stacktop
+  mov    rcx,QWORD PTR fs:[rax]     ; fetch stacktop
+  mov    rdx,rcx
+  add    rdx,0xffffffffffffffd0     ; allocate
+  mov    QWORD PTR fs:[rax],rdx     ; update stacktop
```

References

- Smashing The Stack For Fun And Profit
- Scraps of notes on remote stack overflow exploitation
- Bypassing StackShield
- Bypassing Safe Stack