Lec02: Format String Vulnerabilities

Taesoo Kim

Outline

- Off-the-shelf defenses:
    1. ASLR: Address Space Layout Randomization
    2. RELRO: Relocation Readonly

Defense 3: ASLR

- Option: -fPIE -pie
  - Make the binary position independent (so randomizable)
  - Randomization on stack/heap/libs is system-wide configuration

```
$ cd lec02-fmtstr/aslr
$ make; ./aslr
stack       : 0x7fff5b81ae6c
main()      : 0x558e2867e149
heap        : 0x558e291a0270
...
> offset (&global-&main)   : 0x2ee7
> offset (&printf-&system) : 0x14020
```

ASLR: Real Entrophy

- In theory, 47-bit in userspace in x86_64 (run ./check-aslr.sh)

Security Implication of ASLR

- ASLR makes the exploitation harder (i.e., first line defense)
- Attackers first have to “leak” code pointers
  - Stack and heap of the program, or libc module
- Less effective in fork()-based programs
  - e.g., Zygote in Android, a thread pool in Apache

Defense 4: RELRO

- Relocation tables containing func. pointers are common attack vectors
  - RELRO makes it read-only instead of resolving them on demand
- PLT (Procedure Linkage Table) and GOT (Global Offset Table)

```
main(){ puts("hello world!\n)"); }

0x080488b6 <main+211>: sub    esp,0xc
0x080488b9 <main+214>: push   0x80489ff
0x080488be <main+219>: call   0x8048540 <puts@plt>
                              ^
                              +--- not puts() in libc!
```

PLT/GOT Internals (First puts() Call)

```
   0x080488be <main+219>   : call    0x8048540 <puts@plt>
    ↓
▶  0x8048540 <puts@plt>    : jmp     dword ptr [_GOT_+40] ---+
   0x8048546 <puts@plt+6> : push    0x38                <---+ (1)
   0x804854b <puts@plt+11>: jmp     0x80484c0
    ↓
   0x80484c0 : push    dword ptr [_GOT_+4]
   0x80484c6 : jmp     dword ptr [0x804a008] <0xf7fe9240>
    ↓
   0xf7fe9240 <_dl_runtime_resolve>  : push   eax
   0xf7fe9241 <_dl_runtime_resolve+1>: push   ecx
   0xf7fe9242 <_dl_runtime_resolve+2>: push   edx
```

PLT/GOT Internals (Second puts() Call)

```
  0x080488be <main+219>  : call   0x8048540 <puts@plt>
   ↓
▶ 0x8048540 <puts@plt>   : jmp    dword ptr [_GOT_+40] --- (2)
   ↓
  0xf7e09930 <puts>:   push   ebp
  0xf7e09931 <puts+1>: mov    ebp,esp
  0xf7e09933 <puts+3>: push   edi
```

Checking Common Defenses

```
$ checksec /usr/bin/ls
[*] '/usr/bin/ls'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Goals and Lessons

- Learn about the format string bugs
- Understand their security implications
- Understand the off-the-shelf mitigation
- Learn them from the real-world examples (i.e., sudo/Linux)

About Format String

- How does printf() know of #arguments passed?
- How do we access the arguments in the function?

```
1) printf("hello: %d", 10);
2) printf("hello: %d/%d", 10, 20);
3) printf("hello: %d/%d/%d", 10, 20, 30);
```

About a “Variadic” Function

```
int sum_up(int count,...) {
  va_list ap;
  int i, sum = 0;

  va_start (ap, count);
  for (i = 0; i < count; i++)
    sum += va_arg (ap, int);

  va_end (ap);
  return sum;
}
```

About a "Variadic" Function

```
va_start (ap, count);
  lea    eax,[ebp+0xc]               // Q1. 0xc?
  mov    DWORD PTR [ebp-0x18],eax

for (i = 0; i < count; i++)
  sum += va_arg (ap, int);

  mov    eax,DWORD PTR [ebp-0x18]
  lea    edx,[eax+0x4]               // Q2. +4?
  mov    DWORD PTR [ebp-0x18],edx
  mov    eax,DWORD PTR [eax]
  add    DWORD PTR [ebp-0x10],eax
  ...
```

Format String: e.g., printf()

- What happen if we miss one format specifier?
- What happen if we miss one argument?

```
// buggy
1) printf("hello: %d/%d[missing]", 10, 20, 30);
2) printf("hello: %d/%d/%d", 10, 20, [missing]);
```

Format String: e.g., printf()

- What does printf() print out? guess?

```
  printf("%d/%d/%d", 10, 20)

(top)
       +----(n)----+
       |           v
  [ra][fmt][10][20][??][..]
           (1) (2) (3) ....
```

Format String Vulnerabilities

- What if attackers can control the format specifier (fmtstr)?
  - Arbitrary read → info leaks (e.g., code pointers)
  - Arbitrary write → control-flow hijacking
  - Bypass many existing mitigation (e.g., DEP, ASLR)

```
printf("%d/%d/%d", 10, 20, 30)
printf(fmtstr, 10, 20, 30)?
```

About Format String Specifiers

- Very complex, versatile (e.g., > 482 lines document)

```
%p: pointer
%s: string
%d: int
%x: hex

Tip 1. positional argument
   %[nth]$p
   (e.g., %2$p = second argument)
   (e.g., printf("%2$d", 10, 20, 30) -> 20)
```

Implication 1: Arbitrary Read

- If fmtbuf locats on the stack (perhaps, one of caller's),
- Then, we can essentially control its argument!

```
  printf(fmtbuf)
  printf("\xaa\xbb\xcc\xdd%3$s")

(top)
       +---(3rd)---+
       |           v fmtbuf
  [ra][fmt][a1][a2][\xaa\xbb\xcc\xdd%3$s]
           (1) (2) (3) ....

                      (1)(2)(3)
  -> printf("...%3$s", _, _, 0xddccbbaa)
```

More About Format String Specifiers

- We can write #chars printed so far!
- By the way, do we need this? any application?

```
printf("1234%n", &len) -> len=4

%n: write #bytes (int)
%hn (short), %hhn (byte)

Tip 2. width parameter
    %10d: print an int on 10-space word
    (e.g., "        10")
```

Write (sth) to an Arbitrary Location

- Similar to the arbitrary read, we can control the arguments!

```
  printf("\xaa\xbb\xcc\xdd%3$n")

(top)
       +---(3rd)---+
       |           v
  [ra][fmt][a1][a2][\xaa\xbb\xcc\xdd%3$n]
           (1) (2) (3) ....

                      (1)(2)(3)
  -> printf("...%3$n", _, _, 0xddccbbaa)
     *0xddccbbaa = 4 (#chars printed so far)
```

Implication 2: Arbitrary Write

- In fact, we can control what to write (see more in the tutorial)!

```
  printf("\xaa\xbb\xcc\xdd%6c%3$n")

(top)
       +---(3rd)---+
       |           v
  [ra][fmt][a1][a2][\xaa\xbb\xcc\xdd%6c%3$n]
           (1) (2) (3) ....

  -> *0xddccbbaa = strlen("\xaa\xbb\xcc\xdd      ") = 10
```

Notes on Arbitrary Writes

- Writing a "pointer" is painful (i.e., printing humongous number of spaces)
- Utilizing %hhn (byte), %hn (short), smaller writes

```
// writing *0xddccbbaa = 0xdeadbeef -> four writes
*(0xddccbbaa+0) = 0xef
*(0xddccbbaa+1) = 0xbe
*(0xddccbbaa+2) = 0xad
*(0xddccbbaa+3) = 0xde

[ABCDXXX]
 0000           = 0xef
  1111          = 0xbe
   2222         = 0xad
    3333        = 0xde
```

Notes on More Advanced Attacks

- Previous security issues assume that the input buffer locates in stack
- The input buffer in heap has similar implications (a.k.a., blind fmtstr)!

```
(top)
                      +---------+ +---------+
                      |         v |         v
     [ra][fmt][ ... ][fp][ ... ][fp][ ... ][fp]
                      I-th        J-th

  - Overwriting to the location in I-th argument (J-th)
  - Referring the written value via the J-th argfument
```

Exercise: Real-world Examples

- Ex1. Linux block device (CVE-2013-2851)
- Ex2. Linux ext3 (CVE-2013-1848)
- Ex3. sudo (CVE-2012-0809)

CVE-2013-2851: Linux block device

```
int dev_set_name(struct device *dev, const char *fmt, ...) {
  va_list vargs;
  int err;

  va_start(vargs, fmt);
  err = kobject_set_name_vargs(&dev->kobj, fmt, vargs);
  va_end(vargs);
  return err;
}

// @register_disk()
dev_set_name(ddev, disk->disk_name);

// @__nbd_ioctl()
kthread_create(nbd_thread, nbd, nbd->disk->disk_name);
```

CVE-2013-1848: Linux ext3

```
void ext3_msg(struct super_block *sb, const char *prefix,
        const char *fmt, ...)
{
  struct va_format vaf;
  va_list args;

  va_start(args, fmt);

  vaf.fmt = fmt;
  vaf.va = &args;

  printk("%sEXT3-fs (%s): %pV\n", prefix, sb->s_id, &vaf);

  va_end(args);
}
```

CVE-2013-1848: Linux ext3

```
// @get_sb_block()
ext3_msg(sb, "error: invalid sb specification: %s", *data);

// @ext3_blkdev_get()
ext3_msg(sb, "error: failed to open journal device %s: %ld",
         __bdevname(dev, b), PTR_ERR(bdev));
```

CVE-2012-0809: sudo

```
void sudo_debug(int level, const char *fmt, ...) {
  va_list ap;
  char *fmt2;

  if (level > debug_level) return;

  /* Backet fmt with program name and a newline
     to make it a single write */
  easprintf(&fmt2, "%s: %s\n", getprogname(), fmt);
  va_start(ap, fmt);
  vfprintf(stderr, fmt2, ap);
  va_end(ap);
  efree(fmt2);
}
```

Mitigation Strategies

1. Non-POSIX compliant (e.g., Windows)
    - Discarding %n
    - Limiting width (e.g., "%.512x" in XP, "%.622496x" in 2000)
2. Dynamic: enabling FORTIFY in gcc (e.g., Ubuntu)
3. Static: code annotation (e.g., Linux)

Defense 5: FORTIFY

- Option: -D_FORTIFY_SOURCE=2
- Ensuring that all positional arguments are used
  - e.g., %2$d is not ok without %1$d
- Ensuring that fmtstr is in the read-only region (when %n)
  - e.g., "%n" should not be in a writable region

```
$ ./fortify-yes %2$d
*** invalid %N$ use detected ***

$ ./fortify-yes %n
*** %n in writable segment detected ***
```

Defense 5: FORTIFY

```
// @lec02-fmtstr/fortify
$ make diff
...
 0000000000001040 <main>:
     1040:  sub    rsp,0x8
-    1044:  mov    rdi,QWORD PTR [rsi+0x8]
-    104a:  call   1030 <printf@plt>

+    1044:  mov    rsi,QWORD PTR [rsi+0x8]
+    1048:  mov    edi,0x1
+    104f:  call   1030 <__printf_chk@plt>
```

__printf_chk()

```c
// glibc/debug/printf_chk.c
int ___printf_chk (int flag, const char *format, ...) {
  va_list ap; int done;

  if (flag > 0)
    stdout->_flags2 |= _IO_FLAGS2_FORTIFY;

  va_start (ap, format);
  done = vfprintf (stdout, format, ap);
  va_end (ap);

  if (flag > 0)
    stdout->_flags2 &= ~_IO_FLAGS2_FORTIFY;

  return done;
}
```

__printf_chk()

- Ensuring that all positional arguments are used
  - e.g., %2$d is not ok without %1$d

```
// @vprintf()
for (cnt = 0; cnt < nargs; ++cnt)
  switch (args_type[cnt])
    ...
    case -1:
      /* Error case.  Not all parameters appear in N$ format
         strings.  We have no way to determine their type.  */
      assert (s->_flags2 & _IO_FLAGS2_FORTIFY);
      __libc_fatal ("*** invalid %N$ use detected ***\n");
    }
```

__printf_chk()

- Ensuring that fmtstr is in the read-only region (when %n)
  - e.g., "%n" should not be in a writable region

```
// @vprintf()
LABEL (form_number):
  if (s->_flags2 & _IO_FLAGS2_FORTIFY) {
    if (! readonly_format) {
      extern int __readonly_area (const void *, size_t);
      readonly_format \
        = __readonly_area (format, ((STR_LEN (format) + 1)
                                    * sizeof (CHAR_T)));
    }
    if (readonly_format < 0)
      __libc_fatal ("*** %n in writable segment detected ***\n");
  }
```

Defense 6: Code Annotation for Compilers

```
// @include/linux/compiler_types.h
#define __printf(a, b) __attribute__((format(printf, a, b)))

extern __printf(2, 3)
int dev_set_name(struct device *, const char *, ...);

extern __printf(3, 4)
void __ext4_msg(struct super_block *, const char *,
                const char *, ...);
```

Defense 6: Code Annotation for Compilers

```
// @lec02-fmtstr/format
$ cc -g -Wformat test.c -o test
> format ‘%d’ expects a matching ‘int’ argument
   dev_set_name(3, "test3: %d %d\n", 1);          /* YES */
                               ~^
> too many arguments for format
   dev_set_name(3, "test4: %d %d\n", 1, 2, 3);    /* YES */
                   ^~~~~~~~~~~~~~~~
> missing $ operand number in format
   dev_set_name(3, "test4: %2$d %d %d\n", 1, 2); /* FALSE */
                   ^~~~~~~~~~~~~~~~~~~~~
> $ operand number used after format without operand number
   dev_set_name(3, "test4: %d %1$d", 1);          /* FALSE */
   ^~~~~~~~~~~~
```

References

- Bypassing ASLR
- Advanced return-into-lib(c) exploits
- Format string vulnerability
- Blind format string attacks
- A Eulogy for Format Strings
- CVE-2013-2851
- CVE-2013-1848
- CVE-2012-0809