Introduction to SW Security

Chap. 11. SW Flaws (Format String)

Spring, 2018

Cho, Seong-je (조성제)

sjcho at dankook.ac.kr

Computer Security & OS Lab, DKU

Many slides taken from Textbooks (there sites), and Web sites

- **Exploiting Format String Vulnerabilities**, scut, 2001
  - **http://www.madchat.fr/coding/c/c.seku/format_string/formatstring.pdf**

- **Textbook**: R. C. Seacord, Secure Coding in C and C++, Second Ed., Addison-Wesley
  - http://ptgmedia.pearsoncmg.com/images/9780321822130/samplepages/0321822137.pdf

- Insecure Programming by example
  - http://community.coresecurity.com/~gera/InsecureProgramming/

Format String Vulnerabilities

See the paper, "Exploiting Format String Vulnerabilities"
http://www.madchat.fr/coding/c/c.seku/format_string/formatstring.pdf

printf();

```
printf("Color %s, number1 %d, number2 %05d, hex %#x, float %5.2f, unsigned value %u.\n",
                "red", 123456, 89, 255, 3.14159, 250);
```

```
Color red, number1 123456, number2 00089, hex 0xff, float 3.14, unsigned value 250.
```

%[*parameter*][*flags*][*width*][.*precision*][*length*]*type*

- **Flags**
  - +, space, -, # or **0**

  e.g.) printf("%2d", 3) results in " 3", while printf("%02d", 3) results in "03".
- **Length**
  - **hh**: For integer types, causes printf to expect an int-sized integer argument which was promoted from a char.
  - **h**: For integer types, causes printf to expect an int-sized integer argument which was promoted from a short.
- **Type**

  %n: Print nothing, but write number of characters successfully written so far into an integer pointer parameter.

  %p: void * (pointer to void) in an implementation-defined format.

What is the format string vulnerability

- A correct implementation

```
printf (“%s”, “hello”);
printf(“%x %x”,0xffffffff, 0x00000000);
```

- A wrong implementation

```
char buf[MAX_BUF];

printf(argv[1]);
printf(buf);
printf(“%x %x”);
printf (“%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s”);
```

Format String Problems

- The %x specifier enabled you read the stack, four bytes at a time
- Is the app 32- or 64-bit?
- ASLR is turned off

  (Address Space Layout Randomization)

```
/* fmt_bug1.c *
#include <stdio.h>

int main(int argc, char * argv[]) {
  if (argc > 1)  printf (argv[1]);

  printf("\n");
  return 0;
}
```

```
[root@redhat]# ./fmt_bug1 "%x %x"
bffffb88 400309cb
[root@redhat]# ./fmt_bug1 %p
0xbffffb88
[root@redhat]# ./fmt_bug1 %p
0xbffffb88
[root@redhat]# ./fmt_bug1 "%x %x %x"
bffffb88 400309cb 2
$ ./fmt_bug1 %p
00000000006A6790
```

Format string is not a string literal (potentially insecure)

```
/* fmt_bug2.c        */

int main() {
   char  format [32];

   strcpy (format, "%08x, %08x, %08x, %08x");
   printf (format, 1, 2, 3);
}
```

```
8048407:    e8 34 ff ff ff    call   8048340
<_init+0x80>
 804840c:    83 c4 08          add    $0x8,%esp
 804840f:    6a 03             push   $0x3
 8048411:    6a 02             push   $0x2
 8048413:    6a 01             push   $0x1
 8048415:    8d 45 e0          lea
0xffffffe0(%ebp),%eax
 8048418:    50                push   %eax
 8048419:    e8 12 ff ff ff    call   8048330
<_init+0x70>
```

```
[root@redhat]# ./fmt_bug2
00000001, 00000002, 00000003, 78383025[root@redhat]#
```

Format String Vulnerabilities

- %n specifier writes the number of characters that should have been written so far into the address of the variable you gave as the corresponding argument

```
unsigned int bytes;

printf ("%s%n₩n", argv[1], &bytes);
printf ("Your input was %d characters long₩n", bytes);
```

```
$ ./format_bug2 "Some random input"
Some random input
Your input was 17 characters long
```

The use of %n format specifier in C

```
/* fmt_eg3.c        */

main( ) {
        char c;
        short s;
        int i, j;
        long l;
        long long ll;

        printf("hello %hhn\n", &c);
        printf("hello %hn\n", &s);
        printf("hello %n\n", &i);
        printf("hello %ln\n", &l);
        printf("hello %lln\n", &ll);
        printf("c=%c, s=%d, i=%d, l=%ld, ll=%ld\n",
                        c, s, i, l, ll);
}
```

- Length

hh: For integer types, causes printf to expect an int-sized integer argument which was promoted from a char.

l: For integer types, causes printf to expect a long-sized integer argument.

ll: For integer types, causes printf to expect a long long-sized integer argument.

```
[root@redhat]# ./fmt_eg3
hello
hello
hello
hello
hello
c=, s=6, i=6, l=6, ll=6
```

The use of %n format specifier in C

```
/* fmt_eg4.c        */

main( ) {
        short s;
        int i, j;
        long l;
        long long ll;

        printf("hello1 %hn\n", &s);
        printf("hello12 %n\n", &i);
        printf("hello123 %n\n", &j);
        printf("hello1234 %ln\n", &l);
        printf("hello12345 %lln\n", &ll);
        printf("s=%d, i=%d, j=%d, l=%ld, ll=%ld\n",
                s, i, j, l, ll);
}
```

%n: Print nothing, but write number of characters successfully written so far into an integer pointer parameter.

```
[root@redhat]# ./fmt_eg4
hello1
hello12
hello123
hello1234
hello12345
s=7, i=8, j=9, l=10, ll=11
```

The use of %n format specifier

```
/* fmt_eg5.c        */

main() {
        int i=0;
        int j=0, k;

        printf("%10u%n", 1, &i);
        printf("%100u%n", 1, &j);
        printf("%12345u%n", 1, &k);
        printf("\n");
        printf("i=%d, j=%d, k=%d\n", i, j, k);
}
```

%[*width*][.*precision*][*length*]*type*

width: specifies a *minimum* number of characters to output, and is typically used to pad fixed-width fields in tabulated output, where the fields would otherwise be smaller, although it does not cause truncation of oversized fields. A leading zero in the width value is interpreted as the zero-padding flag mentioned above.

%u: Print decimal unsigned int.

```
[root@redhat]# ./fmt_eg5
         1
        1
                                  1
i=10, j=100, k=12345
```

The use of %n format specifier

```
/* fmt_eg6.c        */

main() {
      unsigned char foo[4];
      int i;

      memset(foo, '\x41', 4);
      for (i=0; i<4; i++)
            printf("%4x, ", foo[i]);
      printf("\n");

      printf("%16u%n", 1, &foo[0]);
      printf("%32u%n", 1, &foo[1]);
      printf("%64u%n", 1, &foo[2]);
      printf("%128u%n", 1, &foo[3]);

      printf("\n");
      for (i=0; i<4; i++)
            printf("%x, ", foo[i]);
      printf("\n");
}
```

- can write number of characters successfully written so far into an character pointer parameter.

```
[root@redhat]# ./fmt_eg6
 41,  41,  41,  41,
                1                1
                                 1

   1
10, 20, 40, 80,
```

The use of %n format specifier

```
/* fmt_eg6.7        */

main()
{
      unsigned char foo[2];

      printf("foo = %x, %x\n", foo[0], foo[1]);
      printf("%16u%n %32u%n", 1, &foo[0], 1, &foo[1]);
      printf("\n");
      printf("foo = %x, %x\n", foo[0], foo[1]);
}
```

```
[root@redhat FormatString]# ./fmt_eg7
foo = 4, 8
              1                          1
foo = 10, 31
```

Prevention of format string attack

- Many compilers can statically check format strings and produce warnings for dangerous or suspect formats.
  - In the GNU Compiler Collection, the relevant compiler flags are, *-Wall,-Wformat, -Wno-format-extra-args, -Wformat-security, -Wformat-nonliteral*, and -*Wformat=2*.[8]

<source: Options to Request or Suppress Warnings>

http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Warning-Options.html#Warning-Options

  - Most of these are only useful for detecting bad format strings that are known at compile-time.
  - If the format string may come from the user or from a source external to the application, the application must validate the format string before using it

Summary

- Format string problems
  - %x
  - %n
  - A kind of Input validation errors
  - You can get more information at http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf
  - *scanf()* ?

- For more details, please read "*Exploiting Format String Vulnerabilities*" via http://www.madchat.fr/coding/c/c.seku/format_string/formatstring.pdf