# PWN College

## Session 16

Atousa Ahsani

References: https://pwn.college/, https://guyinatuxedo.github.io/

# Format Strings

**What is a Format String?**

Attacks on Format String Vulnerability

Countermeasures

# Format String

- **What is a format string?**

```
printf ("The magic number is: %d\n", 1911);
```

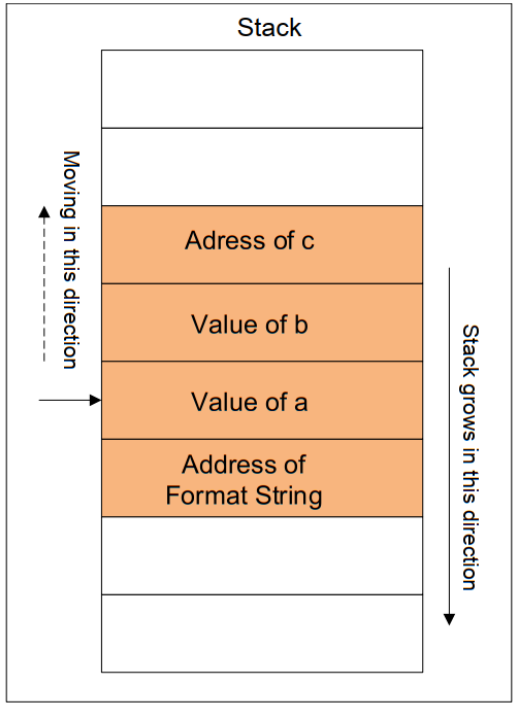
- The text to be printed is "*The magic number is:*", followed by a **format parameter** '*%d*', which is replaced with the parameter (1911) in the output.
- Therefore the output looks like: *The magic number is: 1911*.
- In addition to *%d*, there are several other **format parameters**, each having different meaning.

| Parameter | Meaning | Passed as |
|---|---|---|
| %d | decimal (int) | value |
| %u | unsigned decimal (unsigned int) | value |
| %x | hexadecimal (unsigned int) | value |
| %s | string ((const) (unsigned) char *) | reference |
| %n | number of bytes written so far, (* int) | reference |

# Format String

- **The stack and its role at format strings**
  - The **behavior** of the **format function** is controlled by the **format string**. The function retrieves the parameters requested by the format string from the **stack**.

```
printf ("a has value %d, b has value %d, c is at address: %08x\n",
                                a, b, &c);
```




- The format function now parses the **format string 'A'**, by reading a character a time.
- If it is **not *'%'***, the character is copied to the **output**.
- In case it is, the character behind the ***'%'*** specifies the **type** of **parameter** that should be evaluated.
- The string ***"%%"*** has a special meaning, it is used to **print** the **escape character *'%'*** itself.
- Every other parameter relates to **data** is located on the stack.

# Format String

- **What if there is a miss-match between the format string and the actual arguments?**

```
printf ("a has value %d, b has value %d, c is at address: %08x\n",
                                a, b);
```

  - In the above example, the format string asks for **3 arguments**, but the program actually provides only **two** (i.e. *a* and *b*).

  - **Can this program pass the compiler?**
    - The function *printf*() is defined as function with **variable length of arguments**. Therefore, by looking at the number of arguments, everything looks fine.
    - To find the miss-match, compilers needs to understand how *printf*() works and what the meaning of the **format string** is. However, compilers usually do not do this kind of analysis.
    - Sometimes, the format string is not a constant string, it is generated **during** the execution of the program. Therefore, there is no way for the compiler to find the miss-match in this case.

# Format String

- **What if there is a miss-match between the format string and the actual arguments?**

```
printf ("a has value %d, b has value %d, c is at address: %08x\n",
                                a, b);
```

  - **Can *printf*() detect the miss-match?**
    - The function *printf*() fetches the arguments from the **stack**. If the format string needs 3 arguments, it will fetch **3 data items** from the **stack**. Unless the stack is marked with a boundary, *printf*() does not know that it runs out of the arguments that are provided to it.
    - Since there is no such a marking. *printf*() will continue fetching data from the stack. In a **miss-match** case, it will fetch some data that do not belong to this function call.

# Format Strings

What is a Format String?

**Attacks on Format String Vulnerability**

Countermeasures

# Attacks on Format String Vulnerability

- **Crashing the program**

```
printf ("%s%s%s%s%s%s%s%s%s%s%s");
```

  - For each ***%s***, *printf*() will fetch a **number** from the **stack**, treat this number as an **address**, and print out the **memory contents** pointed by this address as a string, until a **NULL** character (i.e., number 0, not character 0) is encountered.
  - Since the number fetched by *printf*() might not be an address, the memory pointed by this number might **not exist** (i.e. no **physical memory** has been assigned to such an address), and the program will **crash**.
  - It is also possible that the number happens to be a good address, but the **address space** is **protected** (e.g. it is **reserved** for **kernel** memory). In this case, the program will also **crash**.

# Attacks on Format String Vulnerability

- **Viewing the stack**

```
printf ("%08x %08x %08x %08x %08x\n");
```

  - This instructs the printf-function to retrieve **five parameters** from the **stack** and display them as **8-digit padded hexadecimal** numbers.
  - So a possible output may look like:

```
40012980 080628c4 bffff7a4 00000005 08059c04
```

# Attacks on Format String Vulnerability

- **Viewing memory at any location**
  - We have to supply an **address** of the **memory**. However, we cannot change the code; we can only supply the format string.
  - If we use *printf(%s)* without specifying a **memory address**, the target address will be obtained from the **stack** anyway by the *printf*() function. The function maintains an initial **stack pointer**, so it knows the location of the parameters in the stack.
  - ***Observation***: the **format string** is usually located on the **stack**. If we can encode the target address in the **format string**, the target address will be in the **stack**. In the following example, the format string is stored in a **buffer**, which is located on the **stack**.

```
int main(int argc, char *argv[])
{
  char user_input[100];
  ... ... /* other variable definitions and statements */

  scanf("%s", user_input); /* getting a string from user */
  printf(user_input); /* Vulnerable place */

  return 0;
}
```

# Attacks on Format String Vulnerability

- **Viewing memory at any location (contd.)**
  - If we can force the *printf* to obtain the **address** from the **format string** (also on the **stack**), we can control the address.

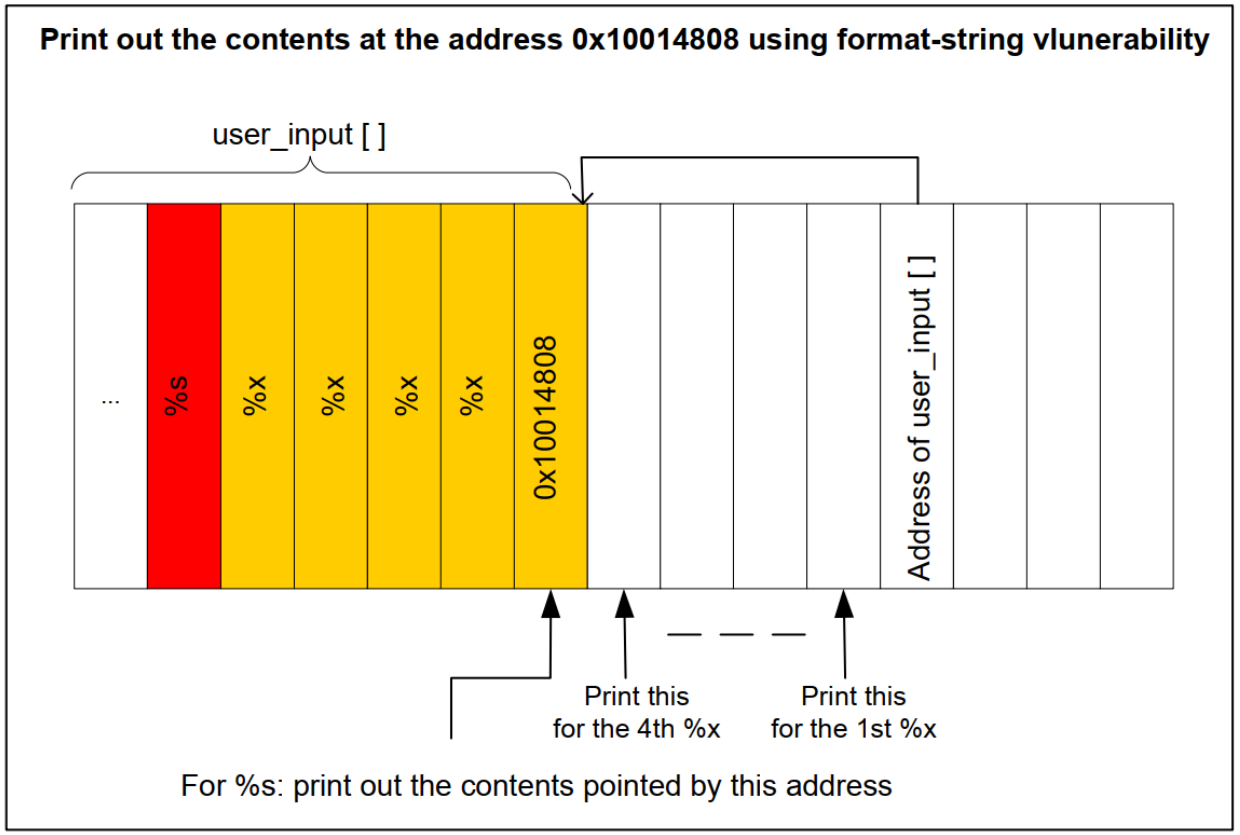
```
printf ("\x10\x01\x48\x08 %x %x %x %x %s");
```

  - *\x10\x01\x48\x08* are the **four** bytes of the **target address**. In **C** language, *\x10* in a string tells the compiler to **put** a **hexadecimal** value ***0x10*** in the current position. The value will take up just **one** byte. Without using **\x**, if we directly put "**10**" in a string, the **ASCII values** of the characters '**1**' and '**0**' will be stored. Their ASCII values are **49** and **48**, respectively.
  - *%x* causes the **stack pointer** to move towards the format string.
  - Here is how the attack works if ***user_input[]*** contains the following **format string**:

```
"\x10\x01\x48\x08 %x %x %x %x %s".
```

# Attacks on Format String Vulnerability

- Viewing memory at any location (contd.)



# Attacks on Format String Vulnerability

- **Viewing memory at any location (contd.)**
  - Basically, we use **four** ***%x*** to move the *printf()*'s **pointer** towards the **address** that we stored in the **format string**. Once we reach the destination, we will give ***%s*** to *print*(), causing it to print out the contents in the memory address *0x10014808*. The function *printf*() will treat the contents as a **string**, and print out the string until reaching the **end** of the string (i.e. 0).
  - The stack space between ***user_input***[] and the **address** passed to the *printf*() function is **not** for *printf*(). However, because of the **format-string** vulnerability in the program, *printf*() considers them as the **arguments** to match with the ***%x*** in the format string.
  - The key challenge in this attack is to figure out the **distance** between the ***user_input***[] and the **address** passed to the *printf*() function. This distance decides **how many** ***%x*** you need to insert into the format string, before giving ***%s***.

# Attacks on Format String Vulnerability

- **Writing an integer to nearly any location in the process memory**
  - ***%n***: The **number** of **characters** written so far is stored into the **integer** indicated by the corresponding argument.

```
int i;
printf ("12345%n", &i);
```

  - It causes *printf*() to write **5** into variable ***i***.
  - Using the same approach as that for **viewing** memory at any **location**, we can cause *printf*() to write an **integer** into any location. Just replace the ***%s*** in the above example with ***%n***, and the contents at the address ***0x10014808*** will be overwritten.
  - Using this attack, attackers can do the following:
    - **Overwrite** important program **flags** that **control access privileges**
    - **Overwrite return addresses** on the **stack**, **function pointers**, etc.
  - However, the **value** written is determined by the **number** of **characters** printed before the ***%n*** is reached. Is it really possible to write **arbitrary** integer values?
    - Use **dummy output characters**. To write a value of **1000**, a simple **padding** of **1000 dummy characters** would do.

# Format Strings

What is a Format String?

Attacks on Format String Vulnerability

Countermeasures

# Countermeasures

- **Address randomization (ASLR)**
  - Just like the countermeasures used to protect against **buffer-overflow** attacks, **address randomization** makes it difficult for the attackers to **find out** what address they want to **read/write**.