SC3010
Computer Security

Lecture 3: Memory Safety Vulnerabilities

Tianwei Zhang

Last Week: Buffer Overflow Vulnerability

Definition: more input are placed into a buffer than the capacity allocated, overwriting other information

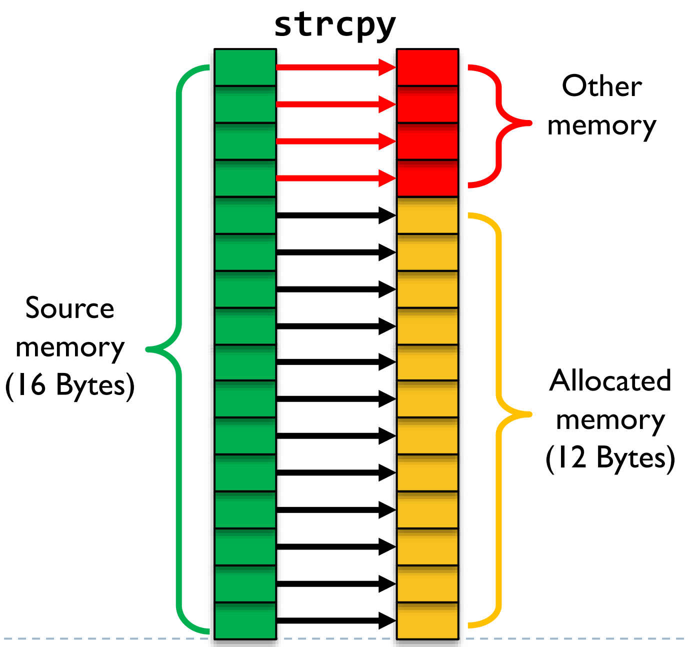
Consequences: overwriting adjacent memory locations could cause

- corruption of program data
- unexpected transfer of control
- memory access violation
- execution of code chosen by attacker

```c
#include <stdio.h>
#include <string.h>

void foo(char *s) {
           char buf[12];
           strcpy(buf,s);
           printf("buf is %s\n",buf);
}

int main(int argc, char* argv[]) {
    foo("Buffer-Overflow!");
    return 0;
}
```



Outline

- **Format String Vulnerabilities**
- **Integer Overflow Vulnerabilities**
- **Scripting Vulnerabilities**

Outline

- **Format String Vulnerabilities**
- Integer Overflow Vulnerabilities
- Scripting Vulnerabilities

printf in C

printf: prints a format string to the standard output (screen).

- Format string: a string with special format specifiers (escape sequences prefixed with `%')
- printf can take more than one argument. The first argument is the format string; the rest consist of values to be substituted for the format specifiers.

Examples.

- `printf("Hello, World");`

  ```
  Hello, World
  ```
- `printf("Year %d", 2014);`

  ```
  Year 2014
  ```
- `printf("The value of pi: %f", 3.14);`

  ```
  The value of pi: 3.140000
  ```
- `printf("The first character in %s is %c", "abc", 'a');`

  ```
  The first character in abc is a
  ```

Format String

Format	Output	Example
d *or* i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	B8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	

A Main Source of Security Problems

Escape sequences are essentially instructions.

- Attack works by injecting escape sequences into format strings.

A vulnerable program

- Attacker controls both escape sequences and arguments in `user_input`.
- The number of arguments should match the number of escape sequences in the format string.
- Mismatch can cause vulnerabilities
- C compiler does not (is not able to) check the mismatch

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
  char user_input[100];
  scanf("%s", user_input);
  printf(user_input);
}
```

More Similar Vulnerable Functions

Functions	Descriptions
`printf`	prints to the 'stdout' stream
`fprintf`	prints to a FILE stream
`sprintf`	prints into a string
`snprintf`	prints into a string with length checking
`vprintf`	prints to 'stdout' from a va_arg structure
`vfprintf`	print to a FILE stream from a va_arg structure
`vsprintf`	prints to a string from a va_arg structure
`vsnprintf`	prints to a string with length checking from a va_arg structure
`syslog`	output to the syslog facility
`err`	output error information
`warn`	output warning information
`verr`	output error information with a va_arg structure
`vwarn`	output warning information with a va_arg structure
	……

Attack 1: Leak Information from Stack

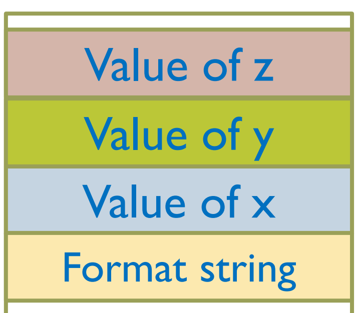
Correct function: `printf("x value: %d, y value: %d, z value: %d", x, y, z);`

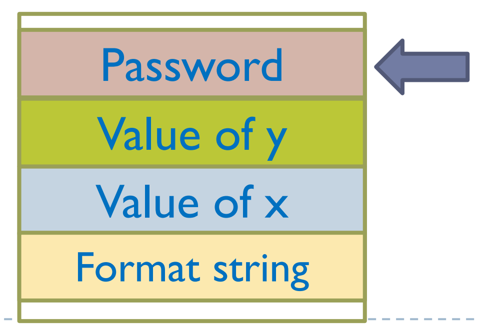
- Four arguments are pushed into the stack as function parameter

Incorrect function: `printf("x value: %d, y value: %d, z value: %d", x, y);`

- The stack does not realize an argument is missing, and will retrieve the unauthorized data from the stack as the argument to print out.
- Data are thus leaked to the attacker

A neat way to view the stack: `printf("%08x %08x %08x %08x %08x");`





Data that do not belong to the user will be printed out

Attack 2: Crash the Program

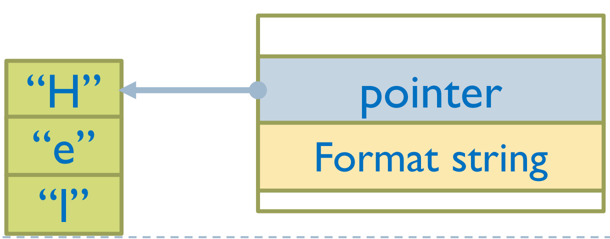
Correct function: `printf(“%s”, “Hello, World”);`

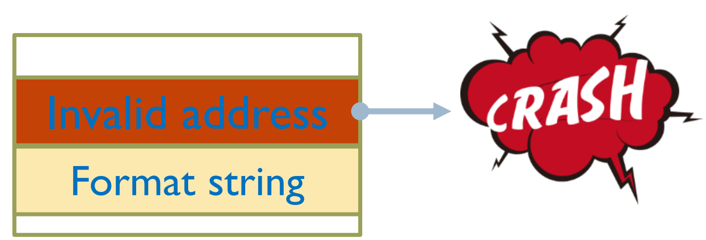
- The pointer of the string is pushed into the stack as function parameter

Incorrect function: `printf(“%s”);`

- The stack does not realize an argument is missing, and will retrieve the data from the stack to print out data at this address.
- This address can be invalidated and program will crash
  - No physical address has been assigned to such address
  - The address is protected (kernel memory)

Increase the crash probability: `printf(“%s%s%s%s%s%s%s%s%s%s%s%s”);`





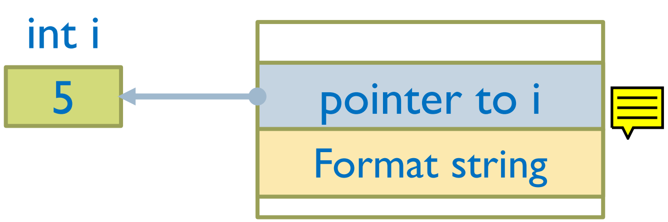
Attack 3: Modify the Memory

Correct function: `printf(“13579%n”, &i);`

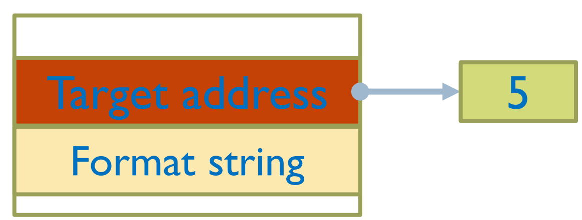
- Store the number of characters written so far (5) into an integer (i)

Incorrect function: `printf(“13579%n”);`

- The stack does not realize an argument is missing, and will retrieve the data from the stack and write 5 into this address.
- Attacker can achieve the following goal:
  - Overwrite important program flags that control access privileges
  - Overwrite return addresses on the stack, function pointers, etc.



if it happens to be a valid address,
if not it will just crash like Attack 2



Summary of Format String Vulnerability

Attacker can abuse the format string %d to cause the violation of ____ property

A. Confidentiality; B. Integrity; C. Availability

Attacker can abuse the format string %s to cause the violation of ____ property

A. Confidentiality; B. Integrity; C. Availability

Attacker can abuse the format string %n to cause the violation of ____ property

A. Confidentiality; B. Integrity; C. Availability

History of Format String Vulnerabilities

Originally noted as a software bug (1989)

- By the fuzz testing work at the University of Wisconsin

Such bugs can be exploited as an attack vector (September 1999)

- **`snprintf`** can accept user-generated data without a format string, making privilege escalation was possible

Security community became aware of its danger (June 2000)

Since then, a lot of format string vulnerabilities have been discovered in different applications.

Application	*Found by*	*Impact*	*years*
wu-ftpd 2.*	security.is	remote root	> 6
Linux rpc.statd	security.is	remote root	> 4
IRIX telnetd	LSD	remote root	> 8
Qualcomm Popper 2.53	security.is	remote user	> 3
Apache + PHP3	security.is	remote user	> 2
NLS / locale	CORE SDI	local root	?
screen	Jouko Pynnōnen	local root	> 5
BSD chpass	TESO	local root	?
OpenBSD fstat	ktwo	local root	?

How to Fix Format String Vulnerability

Limit the ability of adversaries to control the format string

- Hard-coded format strings.
- Do not use %n
- Compiler support to match printf arguments with format string

user can key in %.08x %.08x to get the stack info

to enforce

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
  char user_input[100];
  scanf("%s", user_input);
  printf(user_input);
}
```

```
printf("%s\n", user_input);
```

This way, the printf function will treat user_input as a string and print its contents without interpreting it as a format string.

Outline

- Format String Vulnerabilities
- **Integer Overflow Vulnerabilities**
- Scripting Vulnerabilities

Integer Representation

In mathematics integers form an infinite set.

In a computer system, integers are represented in binary.

- The representation of an integer is a binary string of fixed length (precision), so there is only a finite number of "integers".

Signed integers can be represented as 2's complement numbers.

Most Significant Bit (MSB) indicates the sign of the integer

- MSB is 0: positive integer
- MSB is 1: negative integer

Two's Complement

Positive numbers

- MSB is 0
- Rest digits are in normal binary representation

  $0111\ 1111\ (127); 0000\ 0111\ (7)$

Negative numbers

- MSB is 1
- Conversion from 2's Complement:
  - Flip all the bits and add 1:

  $1111\ 1111 \rightarrow 0000\ 0000 \rightarrow 0000\ 0001 \rightarrow -1$

  $1000\ 0000 \rightarrow 0111\ 1111 \rightarrow 1000\ 0000 \rightarrow -128$

- Conversion to 2's complement:
  - Take the binary representation of the positive part, flip all the bits and add 1

  $-1 \rightarrow 0000\ 0001 \rightarrow 1111\ 1110 \rightarrow 1111\ 1111$

  $-128 \rightarrow 1000\ 0000 \rightarrow 0111\ 1111 \rightarrow 1000\ 0000$

Integer Overflow

An integer is increased over its maximal value, or decreased below its minimal value.

- Unsigned overflow: the binary representation cannot represent an integer value.
- Signed overflow: a value is carried over to the sign bit

In mathematics: $a + b > a$ and $a - b < a$ for $b > 0$

- Such obvious facts are no longer true for binary represented integers

Integer overflow is difficult to spot, and can lead to other types of bugs, frequently buffer overflow.

Arithmetic Overflow

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {

    unsigned int u1 = UINT_MAX;
    u1 ++;
    printf("u1 = %u\n", u1);

    unsigned int u2 = 0;
    u2 --;
    printf("u2 = %u\n", u2);

    signed int s1 = INT_MAX;
    s1 ++;
    printf("s1 = %d\n", s1);

    signed int s2 = INT_MIN;
    s2 --;
    printf("s2 = %d\n", s2);
}
```

⇨ 4,294,967,295

⇨ 0

⇨ 4,294,967,295

⇨ 2,147,483,647

⇨ -2,147,483,648

⇨ -2,147,483,648

⇨ 2,147,483,647

Widthness Overflow

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {

    unsigned int l = 0xdeabeef;
    printf(“l = 0x%u\n", l);

    unsigned short s = l;
    printf(“s = 0x%u\n", s);

    unsigned char c = l;
    printf(“c = 0x%u\n", c);

}
```

0xdeadbeef

0xbeef

0xef

Example 1: Bypass Length Checking

OS kernel system-call handler checks string lengths to defend against buffer overruns.

This is to combine 2 strings together, before combining, check their total lengths if they can fit into the buffer given

```
char buf[128];
combine(char *s1, unsigned int len1, char *s2, unsigned int len2) {
  if (len1 + len2 + 1 <= sizeof(buf)) {
   strncpy(buf, s1, len1);
   strncat(buf, s2, len2);
  }
}
```

The following condition will pass the checking

- len1 < sizeof(buf) (small), len2 = UINT_MAX (large)
- len2 + 1 = 0 so **strncpy** and **strncat** will still be executed.

but the buff size is only 128, can never fit UNIT_MAX

large

e.g. len 1 = 128

become small so it will pass the checking

and strncat will concat such a huge number of char which will leak the memory beyond the size of buf

A better length check

→ make sure both is smaller than buf

```
if (len1 <= sizeof(buf) && len2 <= sizeof(buf)
        && (len1 + len2 + 1 <= sizeof(buf)))
```

Example 2: Write to Wrong Mem Location

Consider an array starting at memory location **0xBBBB** (on a 16-bit machine)

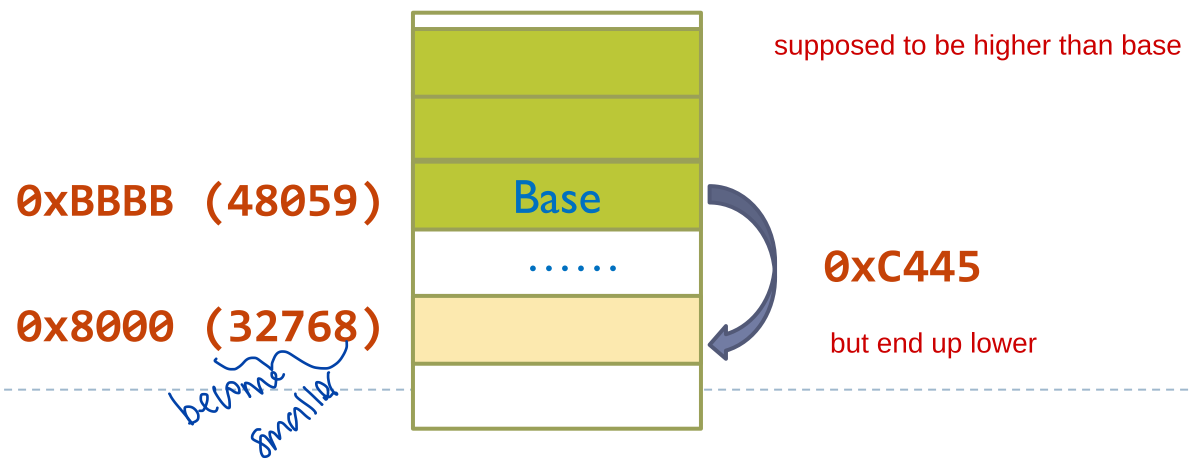
16-bit address

Write to the element at the index of **0xC445**

- $\underbrace{\textbf{0xBBBB}}_{} + \underbrace{\textbf{0xC445}}_{\text{+ index}} = \underbrace{\textbf{0x8000}}_{\text{smaller}}$

The memory location at **0x8000** is overwritten!!

Must check lower bounds for array indices.



Example 3: Truncation Errors

due to the incorrect conversion of long bits to short bits

A bad type conversion can cause widthness overflows

64 bit

```
int func(char *name, long cbBuf) {
    unsigned int bufSize = cbBuf;          truncated / int
    char *buf = (char *)malloc(bufSize);   } small size
smaller
than    if (buf) {
long        memcpy(buf, name, cbBuf);
            ......
            free(buf);
            return 0;
    }
}
```

buf is smaller now but still trying to fit in long bits

Buffer overflow in **memcpy**

- cbBuf is larger than 2^32-1

negative → very large unsigned positive value

Example 4: Signed and Unsigned Vulnerability

also conversion error

Another bad conversion between signed and unsigned integers

signed

```
int func(char *data, int len) {
    char *buf = (char *)malloc(64);

    if (len > 64)
        return 0;

    memcpy(buf, data, len);
}
```

should also check for negative value

will be converted from signed int to unsigned int which changed "-1" to become a very large number and bypass the checking

memcpy will convert int (signed by default) to unsigned (positive) only

Vulnerability:

- `int` is signed, while `memcpy` can only accept unsigned parameter
- `memcpy` will convert `len` from signed integer to unsigned integer
- When `len=-1`, it will be converted to 0xFFFFFFFF, causing buffer overflow.

smaller than 64

1111 1111 ....

Outline

- Format String Vulnerabilities
- Integer Overflow Vulnerabilities
- **Scripting Vulnerabilities**

Scripting Vulnerabilities

Scripting languages

- Construct commands (scripts) from predefined code fragments and user input at runtime
- Script is then passed to another software component where it is executed.
- It is viewed as a domain-specific language for a particular environment.
- It is referred to as very high-level programming languages
- Example:
  - Bash, PowerShell, Perl, PHP, Python, Tcl, Safe-Tcl, JavaScript

Vulnerabilities

- An attacker can hide additional commands in the user input.
- The system will execute the malicious command without any awareness

Example: CGI Script

Common Gateway Interface

- Define a standard way in which information may be passed to and from the browser and server.

Consider a server running the following command

```
cat $file | mail $clientaddress
```

- `$file` and `$clientaddress` are provided by the client.

Normal case:

- A client sets `$file`=hello.txt, and `$clientaddress`=127.0.0.1

```
cat hello.txt | mail 127.0.0.1
```

Compromised Input

- The attacker sets **`$file`** = hello.txt, and `$clientaddress`=127.0.0.1 | rm –rf /
- The command becomes:

```
cat hello.txt | mail 127.0.0.1 | rm -rf /
```

- After mailing the file, all files the script has permission to delete are deleted!

SQL Language

Structured Query Language

- A domain-specific language for database
- Particularly useful for handling structured data

Example

- Get a set of records:

```
SELECT * FROM Accounts WHERE Username= ‘Alice’
```

- Add data to the table:

```
INSERT INTO Accounts (Username, Password) VALUES (‘Alice’, ‘1234’)
```

- Update a set of records:

```
UPDATE Accounts SET Password=‘hello’ WHERE Username= ‘Alice’
```

SQL Injection Vulnerabilities

Consider a database that runs the following SQL commands

```
SELECT * FROM client WHERE name= $name
```

- Requires the user client to provide the input `$name`

Normal case:

- A user sets `$name`=Bob:

```
SELECT * FROM client WHERE name= 'Bob'
```

Compromised input

- The attacker sets `$name` = 'Bob' OR 1=1 --

```
SELECT * FROM client WHERE name= 'Bob' OR 1=1
```

- 1=1 is always true. So the entire client database is selected and displayed, violating the confidentiality.

Real-World SQL Injection Attacks

CardSystems (2006)

- A major credit card processing company. Stealing 263,000 accounts and 43 million credit cards.

7-Eleven (2007)

- Stealing 130 million credit card numbers

Turkish government (2013)

- Breach government website and erase debt to government agencies.

Tesla (2014)

- Breach the website, gain administrative privileges and steal user data.

Cisco (2018)

- Gain shell access.

Fortnite (2019)

- An online game with over 350 million users. Attack can access user data

Cross-Site Scripting (XSS)

Targeting the web applications

- Some websites may require users to provide input, e.g., searching

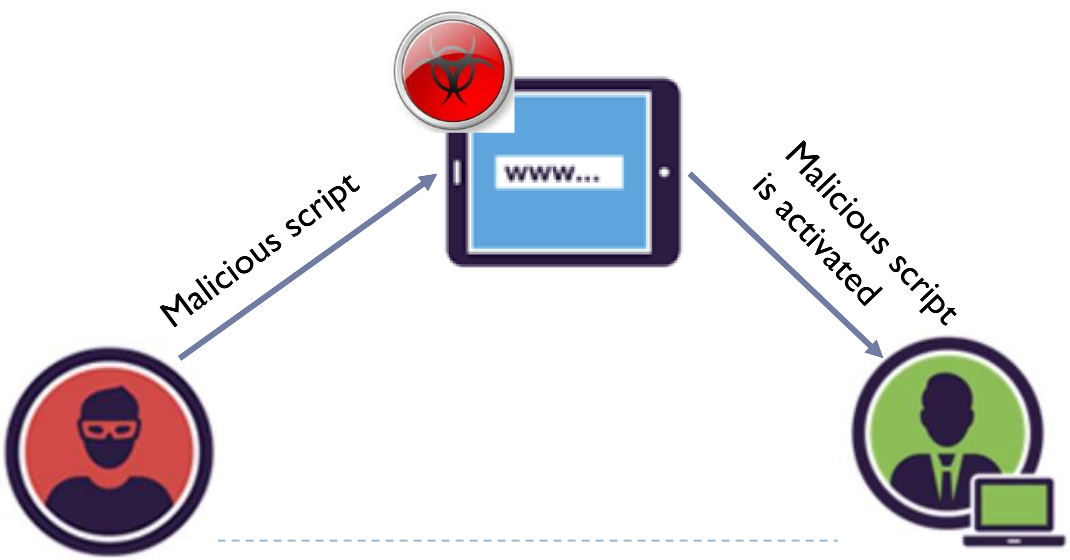
Vulnerabilities

- A malicious user may encode executable content in the input, which can be echoed back in a webpage
- A victim user later visits this web page and his web browser may execute the malicious commands on his computer

Stored XSS Attack (Persistent)

Attack steps

- The attacker discovers a XSS vulnerability in a website
- The attacker embeds malicious commands inside the input and sends it to the website.
- Now the command has been injected to the website.
- A victim browses the website, and the malicious command will run on the victim's computers.



Reflected XSS Attack (Non-persistent)

Attack steps

- The attacker discovers a XSS vulnerability in a website
- The attacker creates a link with malicious commands inside.
- The attacker distributes the link to victims, e.g., via emails
- A victim accidently clicks the link, which actives the malicious commands.