SC3010
Computer Security

Lecture 3: Software Security (II)

Outline

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

Outline

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

printf in C

printf: print 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.	

Security Vulnerability in Format String

Escape sequences are essentially instructions.

- `printf` has no idea how many arguments it actually receives.
- It infers the number of arguments from the format string: number of arguments should match number of escape sequences in the format string.
- What if there is a mismatch?

A vulnerable program

- Users control both escape sequences and arguments in `user_input`.
- An attacker can deliberately cause mismatch between them.

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

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

What potential consequences could this mismatch cause?

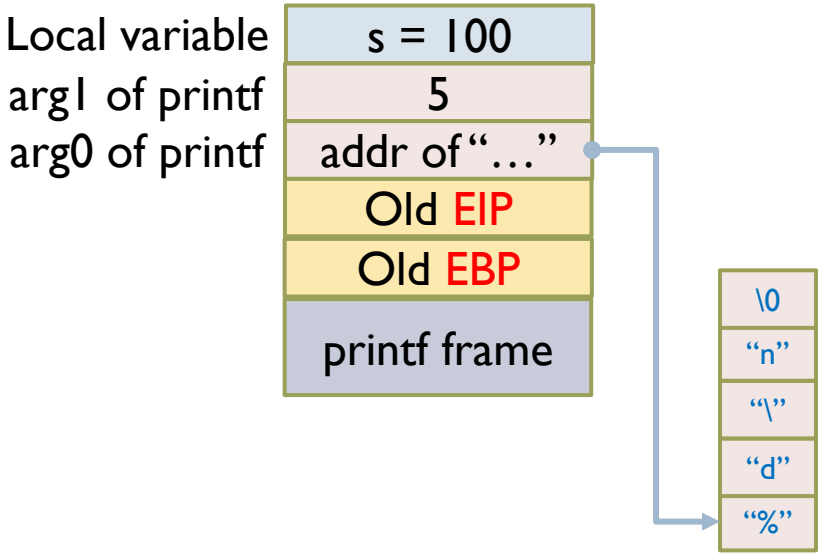
Attack 1: Leak Information from Stack

Correct usage of **printf**

- Two arguments are pushed into the stack as function parameter

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

int main(int argc, char **argv){
    int s = 100;
    printf("%d\n", 5);
    return 0;
}
```



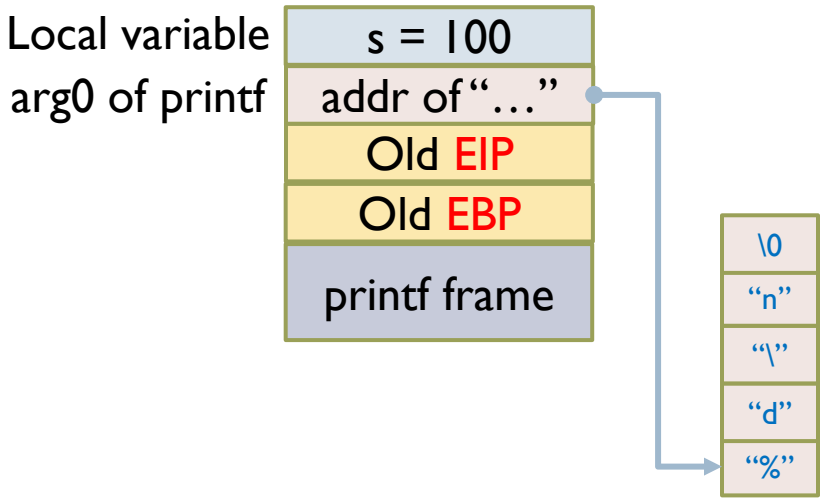
Attack 1: Leak Information from Stack

Incorrect usage of `printf`

- The stack does not realize an argument is missing, and will retrieve the local variable as the argument to print out.
- Data that do not belong to the user are thus leaked to the attacker.
- The attacker can print out any types of data, including integer (**%d**), floating point (**%f**), string (**%s**), address (**%p**)…

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

int main(int argc, char **argv){
    int s = 100;
    printf("%d\n");
    return 0;
}
```



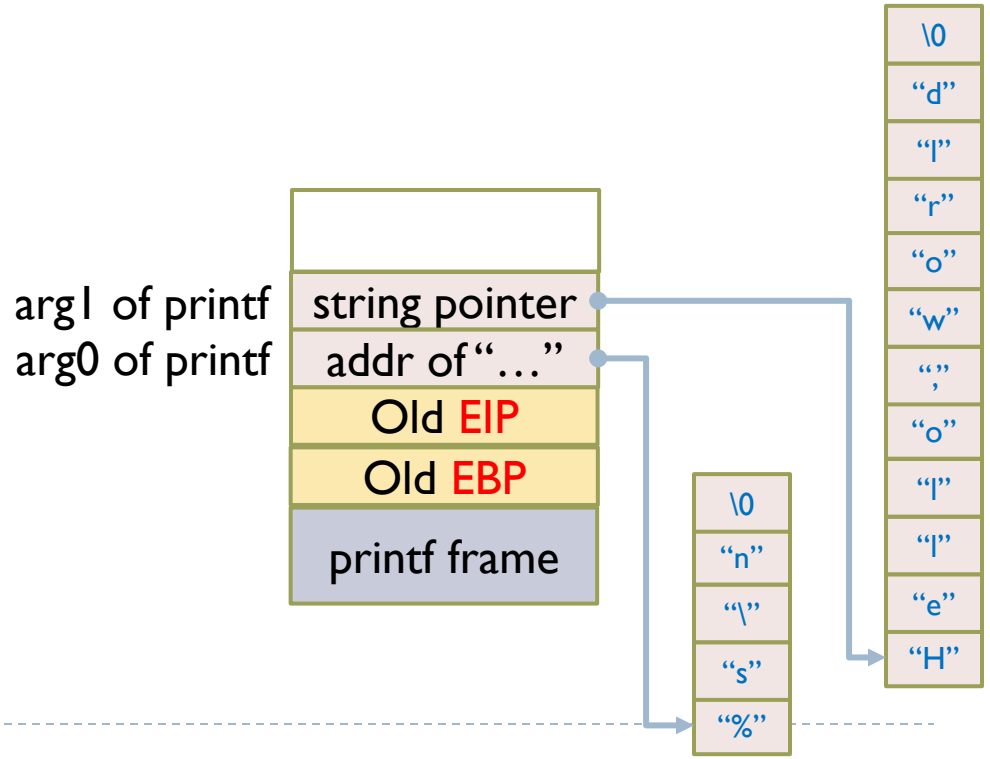
Attack 2: Crash the Program

Correct usage of **printf**

- For format specifier **%s**, a pointer of a string is pushed into the stack as the corresponding function parameter

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

int main(int argc, char **argv){
    printf("%s\n", "hello, world");
    return 0;
}
```



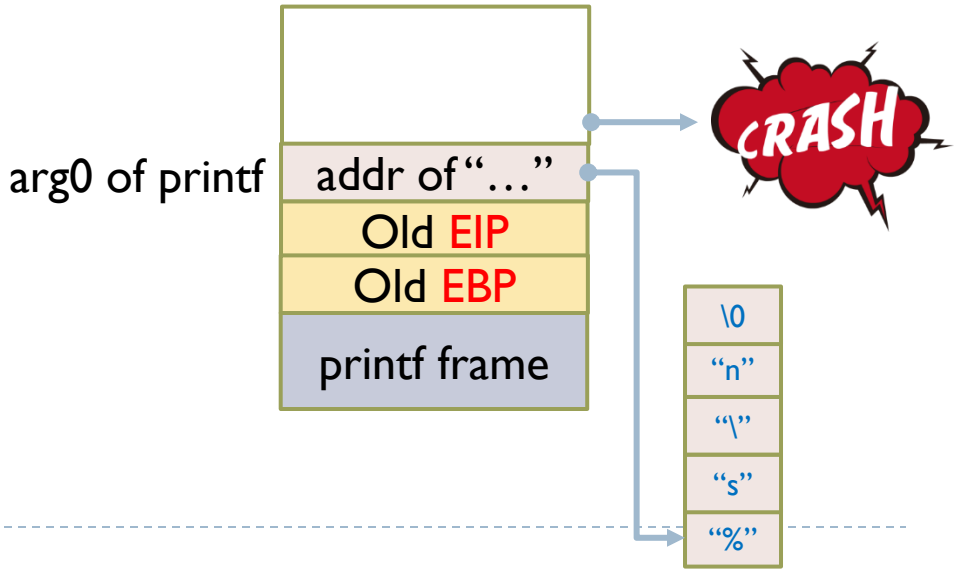
Attack 2: Crash the Program

Incorrect usage of `printf`

- The stack does not realize an argument is missing, and will retrieve other stack values as addresses, and access data there.
- This address can be invalidated, and then the program will crash
  - No physical address is assigned to this address
  - Protected memory, e.g., kernel.
- Can include more `%s` to increase the crash probability

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

int main(int argc, char **argv){
    printf("%s\n");
    return 0;
}
```



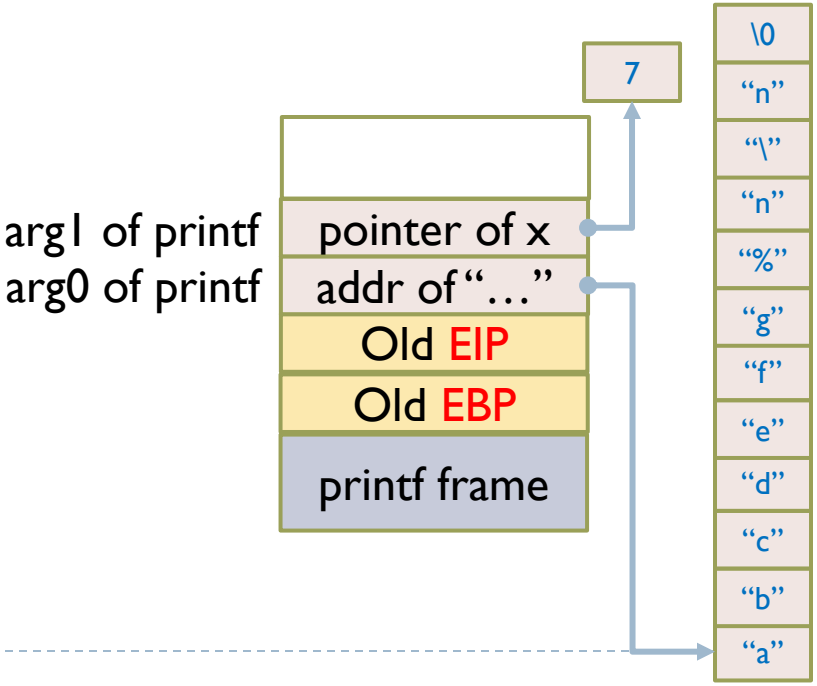
Attack 3: Modify the Memory

Correct usage of `printf`

- For format specifier **%n**, a pointer of a signed integer is pushed into the stack as the corresponding function parameter.
- Store the number of characters written so far into that integer

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

int main(int argc, char **argv){
    int *x = (int *)malloc(sizeof(int));
    printf("abcdefg%n\n",x);
    return 0;
}
```



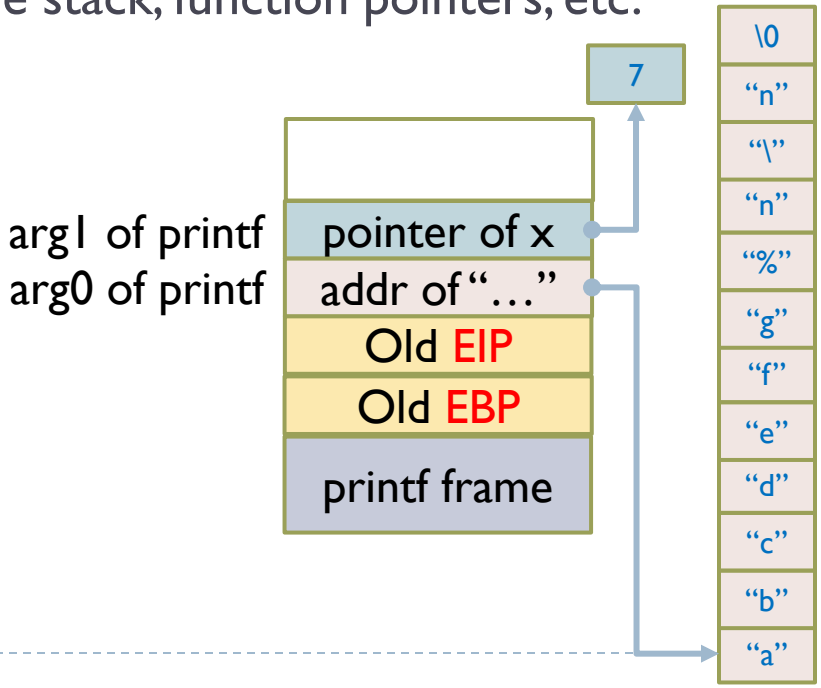
Attack 3: Modify the Memory

Incorrect usage of **printf**

- The stack does not realize an argument is missing, and will retrieve the data from the stack and write 7 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.

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

int main(int argc, char **argv){
    int *x = (int *)malloc(sizeof(int));
    printf("abcdefg%n\n");
    return 0;
}
```



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
……	

History of Format String Vulnerability

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 Pynnönen	local root	> 5
BSD chpass	TESO	local root	?
OpenBSD fstat	ktwo	local root	?

How to Fix Format String Vulnerability

Limit the ability of attackers to control the format string

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

```
#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);
```

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 two's complement: the Most Significant Bit (MSB) indicates the sign of the integer:
  - MSB is 0: positive integer
  - MSB is 1: negative integer.

Integer Overflow

An operation cases its integer operand to increase beyond its maximal value, or decrease below its minimal value. The results are no longer correct.

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

Possible operations that lead to integer overflow.

- Arithmetic operation
- Type conversion.

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

Arithmetic Overflow

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

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

```
#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

Example 1: Bypass Length Checking

Incorrect length checking could lead to integer overflows, and then buffer overflow.

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

char buf[128];
void 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);
  }
}

int main(int argc, char* argv[]) {
    unsigned int len1 = 10;
    unsigned int len2 = UINT_MAX;
    char *s1 = (char *)malloc(len1 * sizeof(char));
    char *s2 = (char *)malloc(len2 * sizeof(char));
    combine(s1, len1, s2, len2);
}
```

Buffer Overflow!

len1 + len2 + 1 = 10 < 128
strncpy and strncat will be executed.

Widthness Overflow

A bad type conversion can cause widthness overflows

```
#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);

}
```

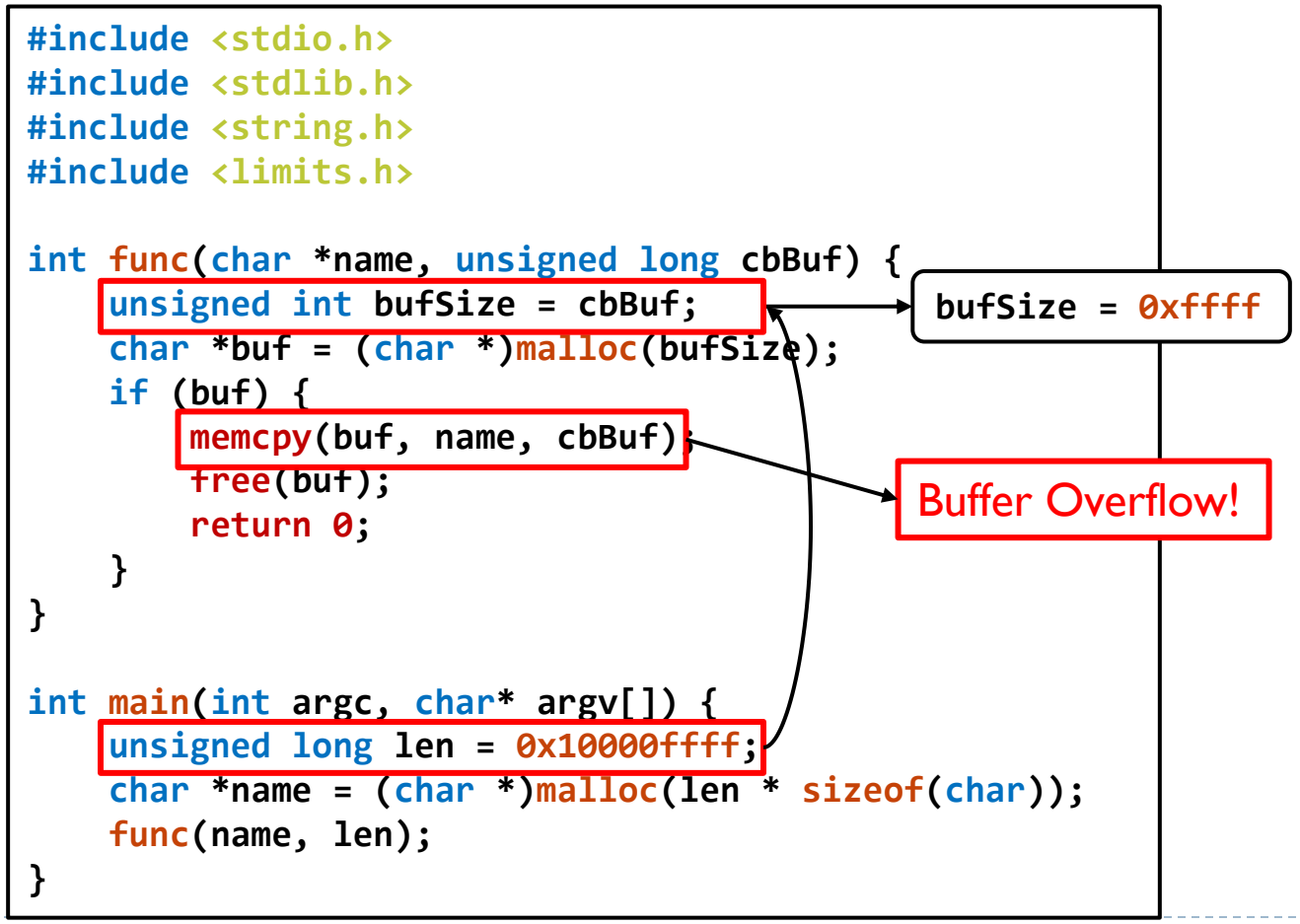
➡ 0xdeadbeef

➡ 0xbeef

➡ 0xef

Example 2: Truncation Errors

Incorrect type conversion could lead to integer overflows, and then buffer overflow.

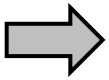


How to Fix Integer Overflow Vulnerability

Be more careful about all the possible consequences of vulnerable operations.

Better length checking

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



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

Safe type conversion:

- Widening conversion: convert from a type of smaller size to that of larger size.

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 1: Command Injection

Consider a server running the following command

- `system`: takes a string as input, spawns shell, and executes the string as command in the shell.

```
void display_file(char* filename) {
  char cmd[512];
  snprintf(cmd, sizeof(cmd), "cat %s", filename);
  system(cmd);
}
```

Normal case:

- A client sets `filename`=hello.txt

```
cat hello.txt
```

Compromised Input:

- The attacker sets `filename` = hello.txt; rm –rf /
- The command becomes:

```
cat hello.txt; rm -rf /
```

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

Defenses against Command Injection

Avoid shell commands

Use more secure APIs

- Python: subprocess.run()
- C: `execve()`

Input inspection

- Sanitization: escape dangerous characters
- Validate and reject malformed input.
- Whitelist: only choose from allowed values

Drop privileges

- Run processes as non-root users.

Example 2: SQL Injection

Structured Query Language (SQL)

- A domain-specific language for managing data in a database

Basic syntax

- Obtain a set of records:

```
SELECT name FROM Accounts
SELECT * FROM Accounts WHERE name= 'Alice'
```

- Add or update data in the table:

```
INSERT INTO Accounts (name, age, password) VALUES ('Charlie', 32, 'efgh')
UPDATE Accounts SET password='hello' WHERE name= 'Alice'
```

- Delete a set of records or the entire table

```
DELETE FROM Accounts WHERE age >= 30
DROP TABLE Accounts
```

- Other syntax characters

  `--` single-line comments

  `;` separate different statements.

Accounts		
name	age	password
Alice	18	1234
Bob	23	5678
Eva	50	abcd

Example 2: SQL Injection

Consider a database that runs the following SQL commands

```
SELECT * FROM Accounts WHERE name= '$name'
```

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

Normal case:

- A user sets `$name`=Bob:

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

Compromised inputs

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

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

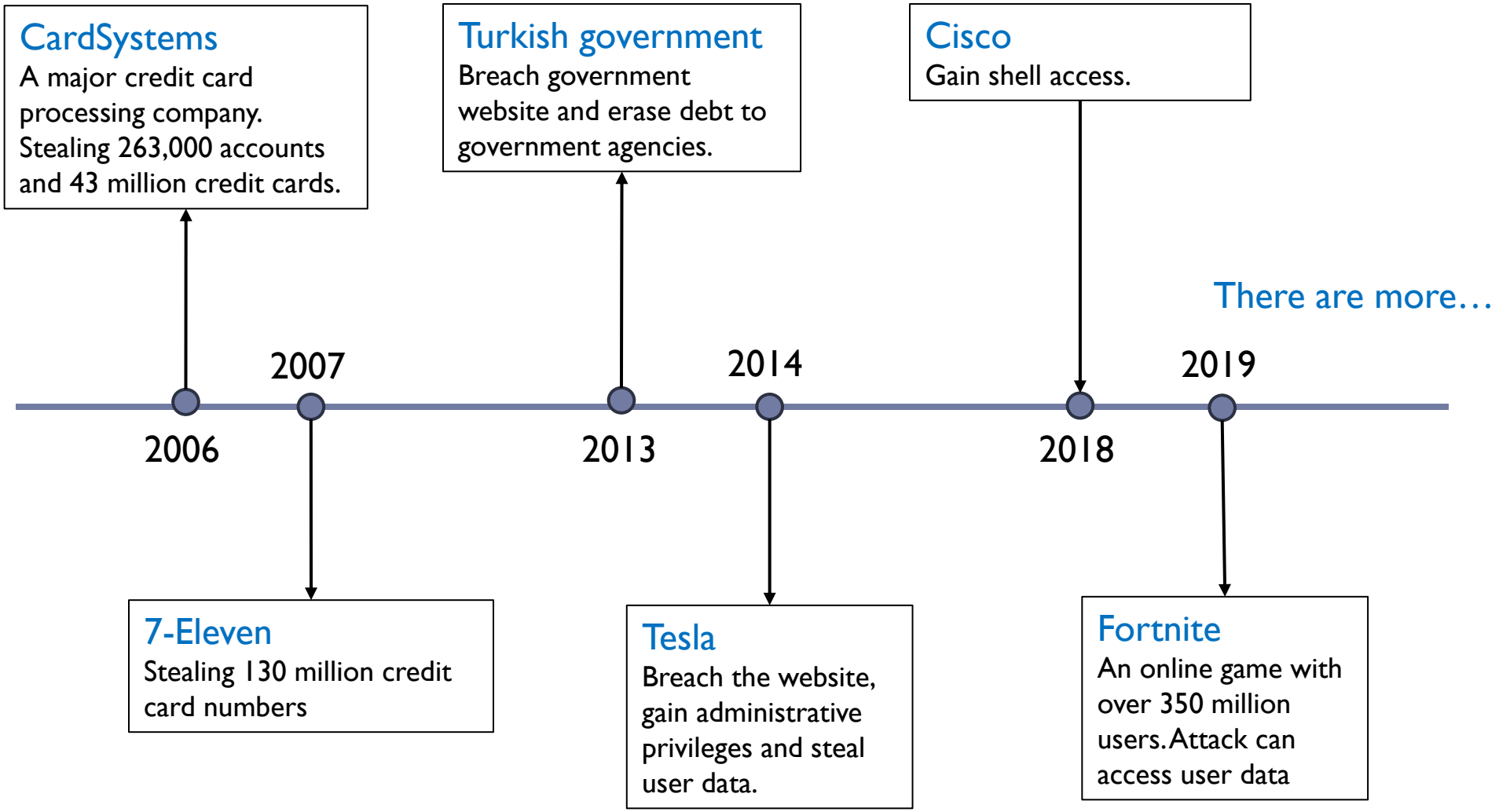
`1=1` is always true. The entire client database is selected and displayed!

- The attacker sets `$name = '; DROP TABLE Accounts --`

```
SELECT * FROM client WHERE name= ''; DROP TABLE ACCOUNTS --'
```

A new statement is injected, which deletes the entire table!

Real-World SQL Injection Attacks



Defenses against SQL Injection

Use parametrized queries

- Ensure that user input is treated as data, not command
- `cursor.execute("SELECT * FROM Accounts WHERE name= ?", (name))`

Object Relational Mapper (ORM)

- Abstract SQL generation and reduce risk of injection

```
class user(DBObject) {
  name =  Column(String(255));
  age =  Column(Integer);
  passsword =  Column(String(255));
}
```

Input inspection

- Sanitization: escape dangerous characters
- Validate and reject malformed input.
- Whitelist: only choose from allowed values

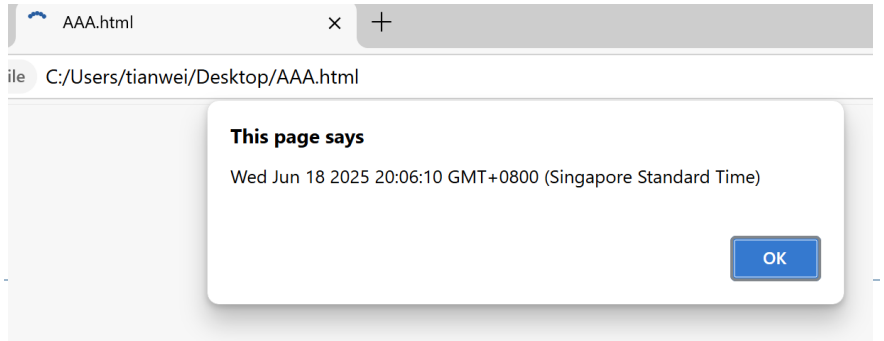
Example 3: Cross-Site Scripting (XSS)

JavaScript

- A programming language for web applications.
- The server sends the JavaScript code to the client, and the browser runs it.
- It makes the website more interactive.

JavaScript can be directly embedded in HTML with `<script>`

```
<html>
  <body>
    <script>alert(Date())</script>
  </body>
</html>
```



Example 3: Cross-Site Scripting (XSS)

Basic idea of XSS

- The attacker injects malicious JavaScript code to a legitimate website
- When victim clients visit the website, the malicious code will be sent to their browsers, and executed on their local computers.
- The malicious code could insert malware to the victims' computers, or collect private information and send it to the remote attacker.

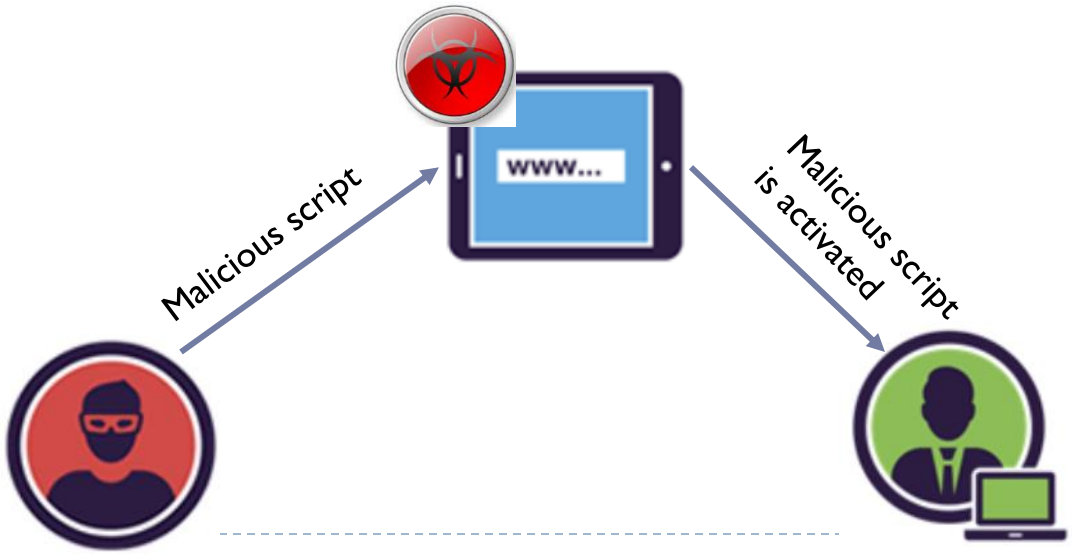
Two types of XSS

- Stored XSS
- Reflected XSS

Stored XSS Attack (Persistent)

Attacker's code is stored persistently on the website

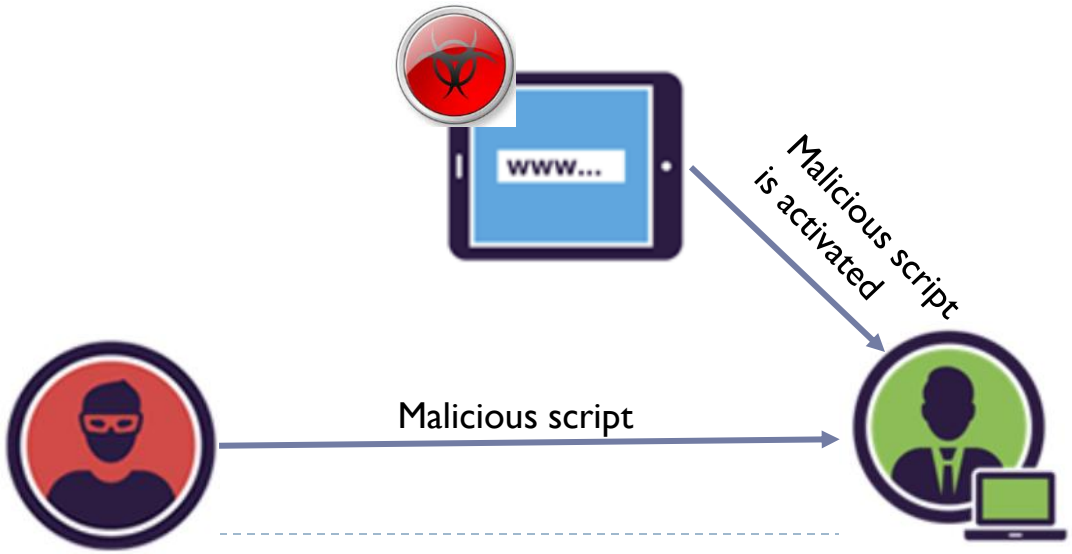
- The attacker discovers a XSS vulnerability in a website
- The attacker embeds malicious commands inside the input and sends it to the website.
- Now the command has been injected to the website.
- A victim browses the website, and the malicious command will run on the victim's computers.



Reflected XSS Attack (Non-persistent)

The attacker tricks the victim to put the code in the request and reflected from the server

- 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, phishing link.
- A victim accidently clicks the link, which actives the malicious commands.



Defenses against XSS

Content Security Policy (CSP)

- Instruct the browser to only use resources loaded from specific places.
- Policies are enforced by the browser.
- Examples of policies
  - Disallow all inline scripts
  - Only allow scripts from specific domains

Input inspection

- Sanitization: escape dangerous characters
- Validate and reject malformed input.