Lecture 19: Other Vulnerabilities

Stephen Huang

Content

1. Integer Overflow

- An integer overflow occurs when you attempt to store inside an integer variable a value larger than the maximum value the variable can hold.
- The C standard defines this situation as undefined behavior (meaning that anything might happen).
- In practice, this usually translates to a wrap of the value if an unsigned integer was used and a change of the sign and value if a signed integer was used.
- Even some managed languages, such as Java and C#, are susceptible to integer overflow errors.

Integer Overflow

- Every integer type has a maximum value.
  – Maximum for 32-bit int is 2,147,483,647
  – The range for 8-bit byte is 0 to 255
- Example

```
unsigned char x = 100;
unsigned char y = 200;
unsigned char z = x + y;
```

`# z == 44` instead of `z == 300`

- Affects most languages
  – even some managed languages, such as Java and C#, are susceptible to integer overflow errors

Integer Overflow Vulnerabilities

- Exploiting integer overflow errors
  - calculating indexes into arrays
  - calculating the amount of space to allocate for a buffer
  - checking whether an overflow could occur
- Example: `// input was provided by the user`

```
const unsigned short SIZE = 10000;
char buffer[SIZE];
unsigned short length = strlen(input);
if (length < SIZE) strcpy(buffer, input);

bool checkOverflow(unsigned short x, unsigned short y)
{
  if (x + y < x) return true;
  return false;
}
```

Does it work?

Incorrect check due to integer promotion

Integer Overflow Vulnerabilities

```
#include <stdio.h>
int main(){
    char a = 30, b = 40, c = 10;
    char d = (a * b) / c;
    printf ("%d ", d);
    return 0;
}
```

Output: `120`

The range for a signed char For most C compilers is = -128 to 127

Integer promotion occurs before the division.

2. Input Validation Weaknesses

- Sources of input
  - User-supplied files and terminal input
  - Command line arguments
  - Environment variables
  - Function calls from other modules
  - Network packets (web applications in detail later)
- Never trust your input
  - Specially crafted input can cause buffer overflows, integer overflows, …
  - Input should always be validated
  - Lack of input validation can lead to software vulnerabilities

Command Injection

- Vulnerable code:

```
void sendEmailToUser(char *username) {
  char buffer[1024];
  sprintf(buffer,
    "mail -s 'Please do not hack us' %s@example.com", username);
  system(buffer);
}
```

- username = "**user**"

```
system("mail -s 'Please do not hack us' user@example.com");
```

- username = "**foo@bar.com ; rm very_important_file ;**"

```
system("mail -s 'Please do not hack us' foo@bar.com ;
                 rm very_important_file ; @example.com")
```

could be any system command, which will be executed by the exploited process

3. Format String Vulnerabilities

- Uncontrolled format strings were discovered to be a vulnerability around 1989, first successful exploits were published around 2000.
- Format string
  - `printf("Security is very %s.", "important");`
  - `printf("The %s is %d!", "answer", 42);`
- Vulnerable functions
  - `sprintf`: writes to buffer
  - `fprintf`: writes to file
  - other members of the `printf` family (e.g., `snprintf`)
  - `printk`: used in the Linux kernel
  - other functions that use format strings (e.g., `syslog`)
- Format placeholders
  - **`%s`** – string (reference)
  - **`%d`** – number (output in decimal format)
  - **`%x`** – number (output in hexadecimal format)
  - …

Safe printf()

```
#include  <stdio.h>
void main(int argc, char **argv)
{      // This line is safe
       printf("%s\n", argv[1]);
}

./example "Hello World %s%s%s%s%s%s"
```

- The `printf` will not interpret the "`%s%s%s%s%s%s`" in the input string, and the output will be: "`Hello World %s%s%s%s%s%s`"

Vulnerable printf()

```
#include  <stdio.h>
void main(int argc, char **argv)
{      // This line is vulnerable
       printf(argv[1]);
}

./example "Hello World %s%s%s%s%s%s"
```

- The `printf` will interpret the `%s%s%s%s%s%s` in the input string as a reference to string pointers, so it will try to interpret every `%s` as a pointer to a string, starting from the location of the buffer (probably on the Stack).
- At some point, it will get to an invalid address, and attempting to access it will cause the program to crash.

Format String Vulnerabilities

```
int main() {
  char string[1024];
  gets(string);
  printf("You wrote: %s\n",
         string);
}


$ ./program
test
You wrote: test

$ ./program
%x %x
You wrote: %x %x
```

```
int main() {
  char string[1024];
  gets(string);
  printf("You wrote: ");
  printf(string);
  printf("\n");
}

$ ./program
test
You wrote: test

$ ./program
%x %x
You wrote: fb7d6460a 7a3b64d06
```

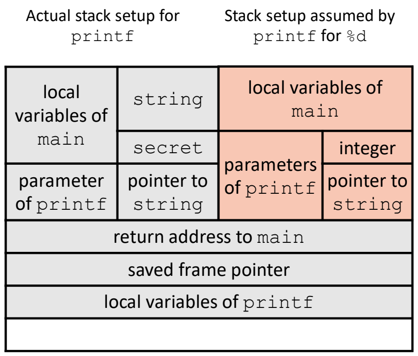
`printf` tries to read parameter values for `%x` from the stack

Exploiting Format Strings

```
int main() {
  char string[1024];
  int secret = 42;

  gets(string);
  printf("You wrote: ");
  printf(string);
  printf("\n");
}
$ ./program
test
You wrote: test

$ ./program
%d
You wrote: 42
```

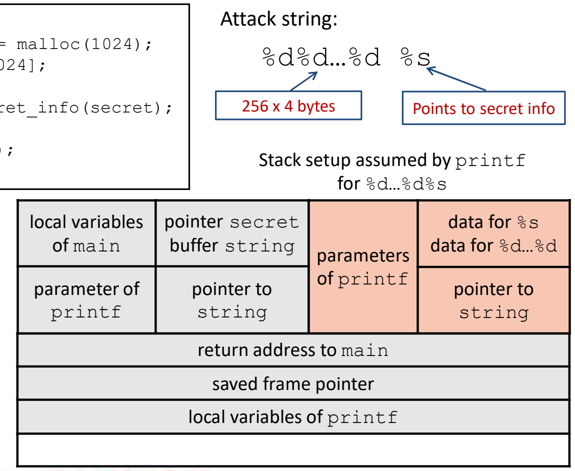


Reading Memory

```
int main() {
  char *secret = malloc(1024);
  char string[1024];

  load_very_secret_info(secret);
  gets(string);
  printf(string);
}
```

Attack string:



alternatively, attacker can supply a memory address for %s within the string → attacker can trick the process into printing the contents of memory at the chosen location

Writing to Memory Using Format

- Special placeholder: **%n**
  - argument must be a pointer to a signed integer, where the number of characters printed so far will be written
  - example: `printf("foobar%n", &x)` writes the value 6 to variable `x`
- Providing a pointer
  - pointer can be part of the malicious string, since it is stored on the stack
  - attacker can use `%x` or `%d` placeholders to reach it
- Controlling the value
  - attacker can control the length of the string
- Arbitrary code execution
  - embed shellcode in the string
  - overwrite return address so that it points to the shellcode

Arbitrary Code Execution

```
int func() {
  char string[1024];
  gets(string);
  printf(string);
}
```

Points to the location of the return address on the stack

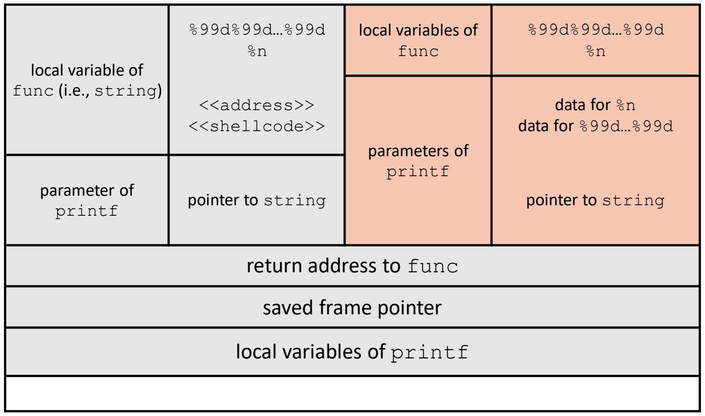
- Attack string:

`%99d%99d…%99d %n <<address>><<shellcode>>`

Two purposes:
- going upwards in the stack until we reach `<<address>>`
- controlling the length of the string → value written to the return address

attacker can overwrite the return address with a value that points to the `<<shellcode>>`

Arbitrary Code Execution



Format String Vulnerabilities

- CVE-2013-1848
  - "fs/ext3/super.c in the Linux kernel before 3.8.4 uses incorrect arguments to functions in certain circumstances related to printk input, which allows local users to conduct format-string attacks and possibly gain privileges via a crafted application."
  - local users may be able to gain superuser (i.e., root) privileges
- CVE-2016-4071
  - "Format string vulnerability in the php_snmp_error function in ext/snmp/snmp.c in PHP before 5.5.34, 5.6.x before 5.6.20, and 7.x before 7.0.5 allows remote attackers to execute arbitrary code via format string specifiers in an SNMP::get call."
- Attacks may combine vulnerabilities
  - first, compromise a process with restricted privileges (e.g., webserver)
  - then, use privilege escalation to gain root access

Malicious Filename as Input

- Example:
  - CVE-2012-0809: "Format string vulnerability in the sudo_debug function in Sudo 1.8.0 through 1.8.3p1 allows local users to execute arbitrary code via format string sequences in the program name for sudo."
  - for debugging purposes, the name of the sudo program (e.g., `/usr/bin/sudo`) is used as part of the format string passed to fprintf(…)
- Exploitation
  - user creates symbolic link to sudo:
    `ln -s /usr/bin/sudo ./<<maliciouscode>><<address>>%x%x%n`
  - user executes sudo (with debugging enabled) using the symbolic link:
    `./<<malicious code>><<address>>%x%x%n -D9`
- Note that the exploit does not require the user to be on the sudoers list. The sudoers file is a file Linux and Unix administrators use to allocate system rights to system users.

4. Race Conditions

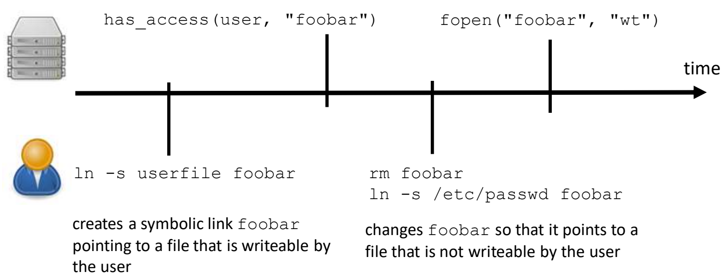
- Race condition
  - When results depend on the sequence or timing of uncontrollable events.
  - for example, when software output depends on how the OS schedules the execution of multiple processes or threads.
- Typically happens when interacting with
  - memory shared by multiple processes (or threads),
  - file system, or
  - signals and other inter-process communication mechanisms.
- Race condition bugs and errors
  - happen when events do not occur in the intended order
  - typically, very difficult to reproduce and debug

Race Condition Vulnerability

- Process with superuser privileges:

```
…
if (has_access(user, filename)) {
  FILE *fout = fopen(filename, "wt");
  write_to_file(fout, data_from_user);
}
…
```

- takes as input from the user a filename and some data
- if the user has access to the file, the process writes data to it



Exploiting a Race Condition

- Challenge (for the attacker): time between checking access and opening the file is very short
- Attack approaches
  - try multiple times (if possible)
  - slow down the target process
    - increase computational load on the machine
    - computational complexity attacks
- Attack techniques for file paths
  - deeply nested directories:
    `filename = "this/is/deeply/nested/.../keep/going/.../wait/for/it/.../almost/there/.../finally/`**`target_file`**`"`
  - chain of symbolic links:

```
ln -s target_file link1; ln -s link1 link2; …; ln -s linkN filename
```


Preventing Race Conditions

- Time of check to time of use
  - we cannot allow any changes in this interval
  - trying to make it short is not enough
- Prevention techniques
  - work with file descriptors instead of filenames
  - rely on filesystem access checks
  - be careful with directories that are writable by everyone (e.g., /tmp/)
  - lock resources (files, databases, etc.)
  - look out for non-atomic operations (e.g., num++)
  - synchronization (e.g., semaphore, mutex)

Race

- These shared memory accesses may happen concurrently
  - if two different processes execute on separate processors, or
  - asynchronously, when a thread sleeps for some time.
- A common cause of race conditions is when a program uses static variables.
- When accessing shared memory locations without appropriate synchronization, ensuring that only one line can utilize them at a time is critical.
- A race condition can occur while accessing a file: the adversary can trick the system by replacing a file with their version and cause the system to read the malicious file.

Race Condition Vulnerability

- CVE-2014-0196
  - "The n_tty_write function in drivers/tty/n_tty.c in the Linux kernel through 3.14.3 does not properly manage tty driver access in the "LECHO & !OPOST" case, which allows local users to cause a denial of service (memory corruption and system crash) or gain privileges by triggering a race condition involving read and write operations with long strings."

5. Buffer Overflow Countermeasures

Compile-time hardening new software	**Run-time protecting existing software**
• programming languages • safe functions and libraries • compiler extensions	• executable space protection • address space layout randomization

unpredictability

5.1 Compile-Time Prevention

Programming Languages

- Programming languages and platforms that do not allow direct memory access typically prevent buffer overflows
  - C# and other managed .NET languages
  - Java, Python, PHP, Perl
- However,
  - safety and security may come at the cost of lower performance (bounds checking can increase execution time)
  - native code written in other languages is still vulnerable (e.g., through Java Native Interface)
  - be careful with libraries that might rely on native code
  - there might be vulnerabilities in the interpreter/virtual machine of the language (e.g., JVM or Common Language Runtime CLR)

Avoid UnSafe Functions

- Unsafe
  - `strcpy(char *src, char *dst)`
  - `strcat(char *s1, char *s2)`
  - `sprintf(char *str, char* format, …)`
  - `gets(char *str)`
- Less Unsafe
  - `strncpy(char *src, char *dst, size_t n)`
  - `strlcpy(char *src, char *dst, size_t n)`
  - `strncat(char *s1, char *s2, size_t n)`
  - `strlcat(char *s1, char *s2, size_t n)`
  - `snprintf(char *str, size_t n, char* format, …)`
  - `fgets(char *str, int n, FILE *file)`

  `(file can be stdin)`

Use Safe Functions

- Safe user input:

```
char buffer[SIZE];
fgets(buffer, sizeof(buffer), stdin);
```

- Memory copying (`memcpy` and `memmove`)
  – less error-prone since we have to specify the source size
  – `memcpy_s(void *dst, size_t dn, void *src, size_t sn)`

Safe Libraries

- C++ Standard Library
  – String class `std::string`
    - mostly safe:
      `str3 = str1 + str2`
    - however, we can shoot ourselves in the foot with the [] operator:

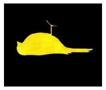
```
string str("ouch");
str[100] = '!';
```

- Array class `std::vector`
  – dynamic-size array implementation
  – but we can use it in an unsafe manner:

```
std::vector<int> array(4);
array[10] = 1;
```


Stack Canary Compiler Extension

In the coal mines, when the bird stopped singing - the miners evacuated immediately.



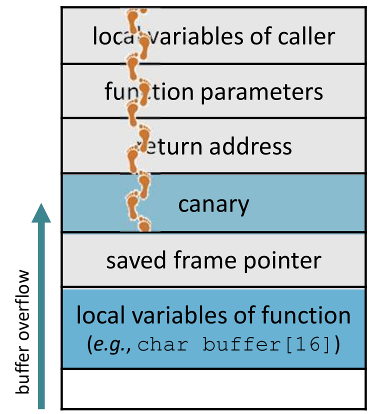
We need a canary to warn us when the return address has been overwritten.

local variables of caller
function parameters
return address
saved frame pointer
local variables of function (*e.g.*, `char buffer[16]`)

Stack Canary Compiler Extension

- At the beginning of the function, a <u>special value</u> is placed on the stack between the local variables and the return address
- Buffer overflows that reach the return address will necessarily overwrite the special value
- Before returning, the function checks if the value is intact

- Stack-based buffer overflow attacks can be detected



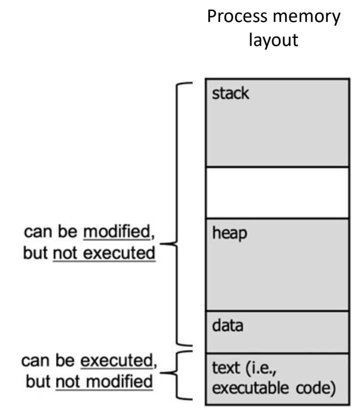
Canary Values & Implementations

- Typical canary values
  - Terminator: contains zero bytes, newline (CR, LF), etc.
    - if input cannot contain one of these values (*e.g.*, strings cannot contain zero bytes), then the canary value is necessarily modified by the exploit
  - Random: random value, typically chosen when the program starts
    - random XOR: canary is the XOR of the random value and the return address
- Implementations
  - Microsoft Visual Studio: `/GS` (Buffer Security Check) option
    - enabled by default, protects control data by creating a copy
  - GCC (GNU Compiler Collection): `-fstack-protector` flag
    - enabled by default, based on a random value

5.2 Run-Time Countermeasures

Executable Space Protection

- Lot of exploits build on injecting and executing malicious code.
- By separating the memory space of a process into executable and modifiable parts, code injection can be prevented.
- *Problem:* Modern computer architectures do not separate code from data.



Hardware-Based Solution NX Bit

- NX (No-eXecute) bit
  - technology in CPUs for separating code from data
  - each page table entry (i.e., data used for managing a part of the memory by the virtual memory system) has an NX bit
    - 0: code can be executed by CPU
    - 1: code cannot be executed by the CPU
  - Hardware-enforced but needs OS support
- Implementations
  - x86: AMD Enhanced Virus Protection or Intel XD (eXecute Disable) bit (Windows and Linux support it)
  - ARM: XN (eXecute Never) bit
- *Limitation:* cannot fully protect programs that create and execute code at runtime (*e.g.*, just-in-time compilers)
  - for example, web browsers that support JavaScript are at risk

Circumventing Executable Space Protection

- An attacker can re-use existing code from the memory space of the process for malicious purposes.
- Return-to-libc attack
  - for most processes, the standard C library is loaded into memory
  - attacker can change the return address of a function to point to the beginning of a function in the C library
  - common target: system function
    - takes as argument a string, and executes it as a system command with the privileges of the process
  - attacker has control over the stack
    → attacker can set up parameters for the C library function

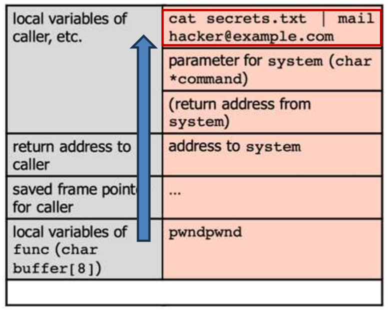
Return-to-libc Attack

Attack string:

`pwndpwnd<…><address to system><…><address to command>cat secrets…`

```
int system(char *command) {
  // standard C library
  // executes command
  ...
}

void func() {
  char buffer[8];
  gets(buffer);
}
```

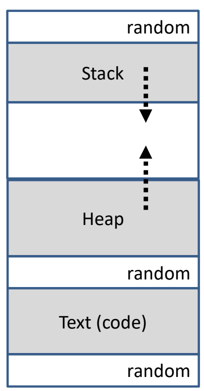


Countermeasure

Address Space Layout Randomization

- In order to reliably jump to an exploited code, the attacker needs to know its address
- Address Space Layout Randomization (ASLR)
  - randomly arrange the positions of the executable, the stack, and the heap in the process's address space
  - may prevent return-to-libc attacks
  - most operating systems (e.g., Windows, Linux) implement some randomization
- Counter-countermeasures
  - information leakage (e.g., printf vulnerability)
  - random guessing

ASLR Process Memory Layout

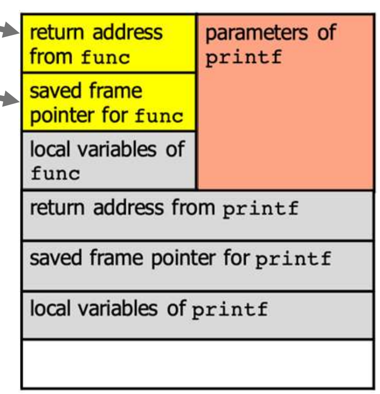


CCM: Information Leakage with printf

- can be used to figure out the address of the executable
- can be used to figure out the address of the stack

```
void func() {
  char test[32];
  gets(test);
  printf("You wrote: ");
  printf(test);
  printf("\n");
}
```

```
$ ./program
%x %x...
You wrote: a4e320ff f0 ...
```



CCM: Random Guessing

- Limitations of ASLR
  - stack and heap cannot be located at any address
    - *example:*
      stack might need to be aligned to 16 bytes and
      heap might need to be aligned to 4096 bytes
  - on a 32-bit system, a brute-force may be viable
- Heap spraying
  - fill up the memory with a certain sequence of bytes
    - *example:* malicious website trying to compromise the client's web browser might fill up the memory using Javascript
  - *example sequences:*
    - shellcode preceded by "NOP slide"
    - "`/////…///bin/sh`" for a return-to-`libc` attack

Next

- Other Vulnerabilities
- **Web Vulnerabilities**
- Malware