Dr. Ammar Haider
Assistant Professor
School of Computing

CS3002 Information Security

Memory Safety Vulnerabilities

Source: Stallings SPP chap 10

Memory Safety

"Consider the example of an application that maintains to do lists for many users.

If I have a to do list with ten items, and I ask for the eleventh item, what should happen? […] In a **memory unsafe language**, unless the programmer explicitly checks which item they're asking for against the length of the list, the item that happens to be at that position in memory is fetched […]. Allowing programs to read past the front or end of a list is called an **out-of-bounds read**. […] if we simply allow someone to ask for the eleventh element of a ten item list, they may get to read the first item out of someone else's list! […]

A closely related spatial vulnerability is an **out-of-bounds write**. In this case imagine we tried to change the eleventh or negative first item in our to do list. Now we are changing someone else's to do list."

https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering/

Buffer Overflow

Out-of-bounds write is commonly known as buffer overflow.

NIST's definition:

> "A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system."

Buffer Overflow: Basics

- Caused by programming error
- Allows more data to be stored than capacity available in a fixed sized buffer
  - buffer can be on stack, heap, global data
- Overwriting adjacent memory locations can cause:
  - corruption of program data
  - bypass security checks
  - unexpected transfer of control
  - memory access violation
  - execution of code chosen by attacker (**control hijacking**)

Example

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
       valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

Load a password e.g., "START" into str1

(a) Basic buffer overflow C code

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

Typical input

Unexpectedly long input

Malicious input

(b) Basic buffer overflow example runs

State of Stack

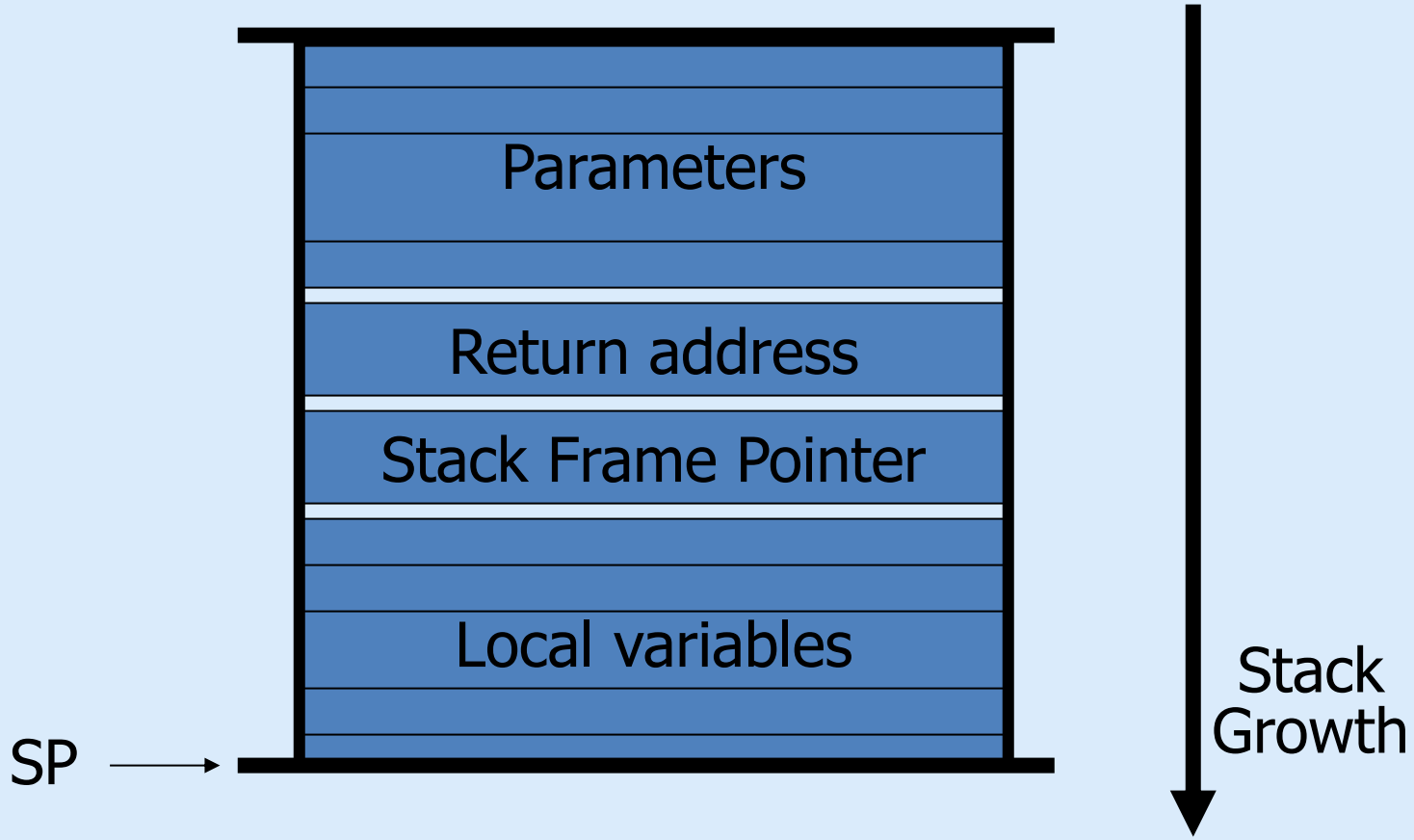
higher addresses

lower addresses

Memory Address	Before gets(str2)	After gets(str2)	Contains value of
. . . .	. . . .	. . . .	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bffffbf0	01000000	01000000	argc
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	return addr
bffffbe8	08fcffbf	08fcffbf	old base ptr
bffffbe4	00000000	01000000	valid
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 V . @	42414449 B A D I	str2[0-3]
. . . .	. . . .	. . . .	

Remember: In general, stack grows backwards (decreasing addresses)

Function calls and stack frames



Buffer Overflow Attacks

- To exploit a buffer overflow an attacker must identify a buffer overflow vulnerability in some program.
  - they can do inspection, trace execution, or use fuzzing tools

- Next, they need to understand how buffer is stored in memory and determine potential for corruption

Programming languages history

- At machine level, all data is an array of bytes
  - interpretation depends on instructions used
- Modern high-level languages (java, c#, python etc.) have a strong notion of data type, valid operations and bounds checking
  - not vulnerable to buffer overflows
  - does incur both compile-time and run-time overhead
- C and related languages have high-level control structures, but allow direct access to memory
  - hence are vulnerable to buffer overflow
  - have a large legacy of widely used, unsafe, and hence vulnerable code

Why use lower level languages

- Systems software has historically been written in lower level languages (C, C++, assembly). These include:
  - operating systems, device drivers, file systems
  - network servers, database servers
  - compilers
  - web browsers
  - command shells and console utilities
  - Internet of Things (IoT) firmwares
- Why?
  - Language adds minimal overheads which means faster performance
  - Raw memory pointers provide more flexibility

* Systems software is in contrast to application software.

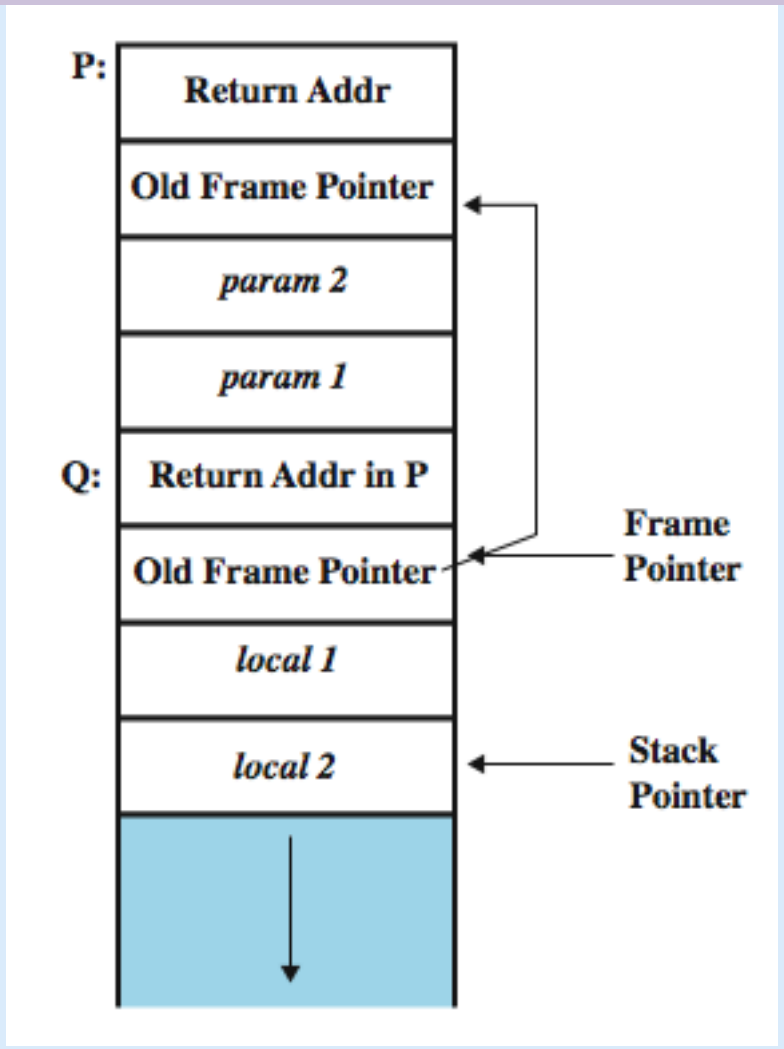
Function calls and stack frames

Stack frame

Calling function: needs a data structure to store the "return" address and parameters to be passed

Called function: needs a place to store its local variables somewhere different for every call

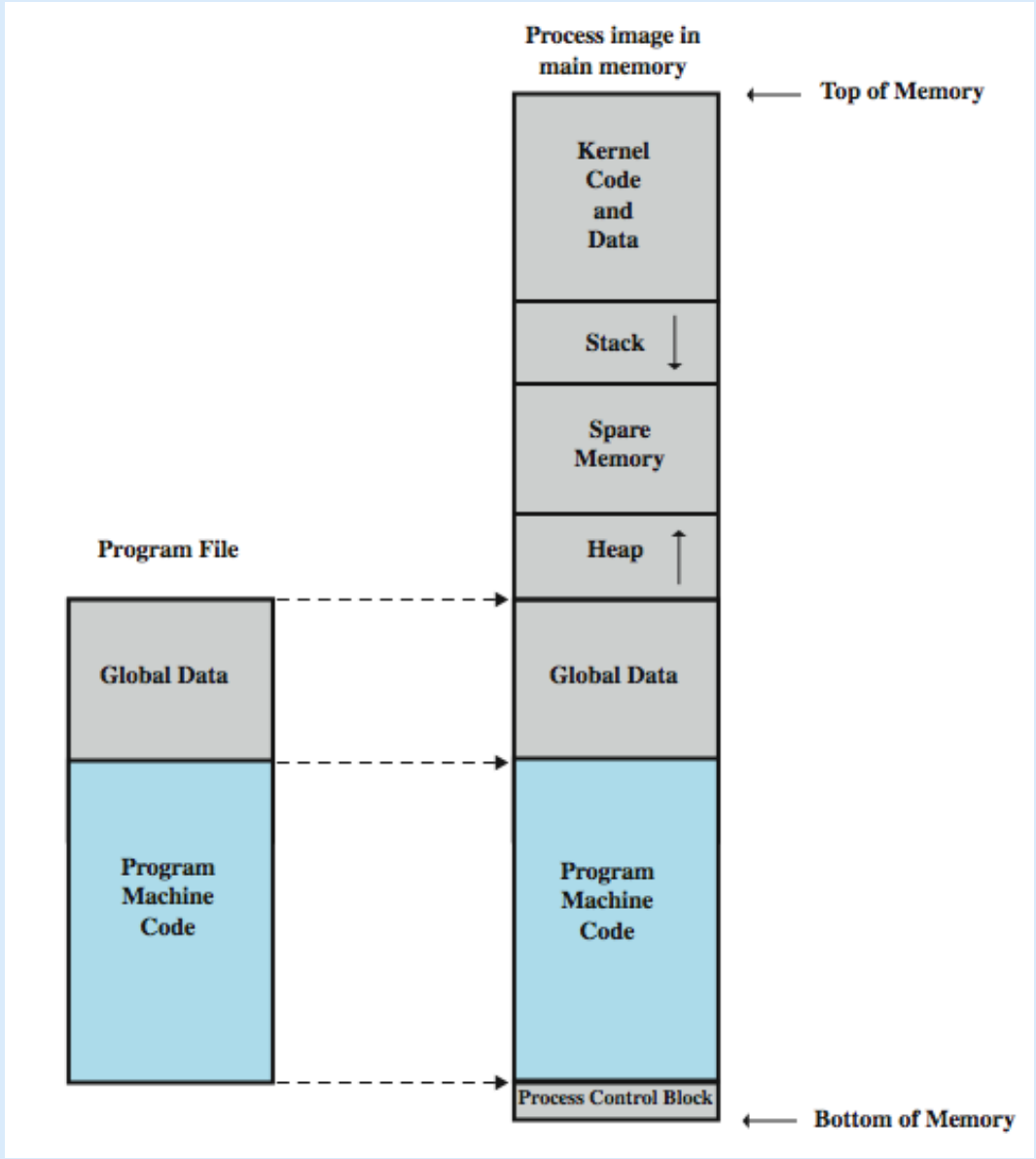
P function calling Q function



Stack Buffer Overflow

- Occurs when buffer is located on stack
  - used by Morris Worm
  - "Smashing the Stack" paper popularized it
- We have local variables below saved frame pointer and return address
  - hence overflow of a local buffer can potentially overwrite these key control items
- Attacker overwrites return address with address of desired code
  - program, system library or loaded in buffer

Programs and Processes



Stack overflow example 2

```
void gctinp(ohar *inp, int siz)
{
    puts("Input value: ");
    fgets(inp, siz, stdin);
    printf("buffer3 getinp read %s\n", inp);
}

void display(char *val)
{
    char tmp[16];
    sprintf(tmp, "read val: %s\n", val);
    puts(tmp);
}

int main(int argc, char *argv[])
{
    char buf[16];
    getinp (buf, sizeof (buf));
    display(buf);
    printf("buffer3 done\n");
}
```

safe input function

sprintf is unsafe — when preparing a string longer than `tmp` size, it will overwrite parts of the stack frame.

Stack overflow example 2

```
$ cc -o buffer3 buffer3.c

$ ./buffer3
Input value:
SAFE
buffer3 getinp read SAFE
read val: SAFE
buffer3 done

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
```

Safe input function: reads only 15 characters

Crash because stack frame of display() was partially corrupted

Common unsafe C functions

Commonly used unsafe functions, and their safer alternatives*

✗ `gets(char *str)` ✓ `fgets(char *str, int n, FILE *stream)`	Read line from standard input into `str`
✗ `strcat(char *dest, char *src)` ✓ `strncat(char *dest, const char *src, size_t n)`	Append contents of string `src` to string `dest`
✗ `strcpy(char *dest, char *src)` ✓ `snprintf` *or non standard* `strlcpy`	Copy contents of string `src` to string `dest`
✗ `sprintf(char *str, char *format, ...)` ✓ `snprintf(char *str, size, char *format, ...)`	Create `str` according to supplied `format`
✗ `vsprintf(char *str, char *format, va_list arg)` ✓ `vsnprintf(char *str, size_t n, char* format, va_list arg)`	Create `str` according to supplied `format` and variables

* Even the safe alternatives require programmer to be careful, and pass correct size value in arguments.

Defenses

- Buffer overflows are widely exploited
- Large amount of vulnerable code in use
  - despite cause and countermeasures known
- Two broad defense approaches
  - compile-time
  - run-time: detect attacks in existing programs
    - can't remove vulnerabilities, but make it harder for attacker to exploit them

Compile-Time Defenses

Choice of programming language (for new projects)

- Modern high level languages (Rust, Swift, Go etc.) apply strong typing and add additional code for bounds checking
  - not vulnerable to buffer overflow
  - compiler enforces permissible operations on variables
  - runtime checking crashes the programs for out-of-bounds read/write
    - crashing is not pretty, but safer than undefined behaviour of C
- These languages do have higher cost in resource use
- …and restrictions on access to hardware
  - so still need some code in C like languages

Compile-Time Defenses

Safe coding techniques

- If using potentially unsafe languages (like C), programmer must explicitly write safe code
  - by design with new code
  - **extensive code review** of existing code, (e.g. OpenBSD)

- Buffer overflow safety a subset of general safe coding techniques

- Allow for graceful failure **(know how things may go wrong)**
  - check for sufficient space in any buffer

Compile-Time Defenses

Language extensions, Safe libraries

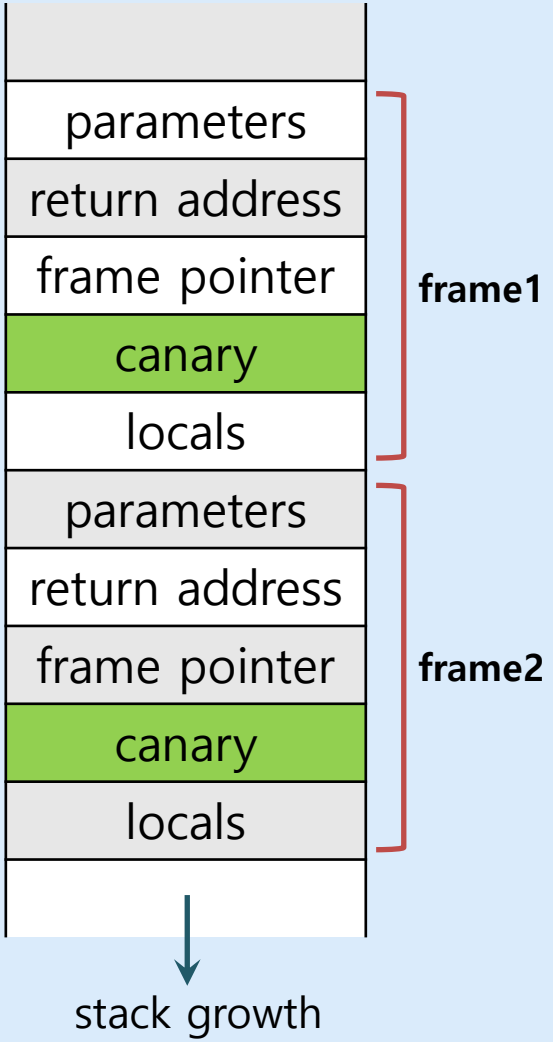
- Proposals for safety extensions to C: memory range checks added by compiler
  - performance penalties
  - must compile programs with special compiler

- Several safer standard library variants
  - new functions, e.g. strlcpy()
  - safer re-implementation of standard functions as a dynamic library, e.g. Libsafe

Compile-Time Defenses

Stack Protection

- Stackgaurd (GCC), Visual C++ /GS flag: add function entry and exit code to check stack for signs of corruption
  - place a randomly chosen value (canary) between local variables and saved frame pointer
  - upon function return, check if canary value has changed. Any buffer overflow attempt would overwrite this value first, before modifying frame pointer and return address
  - abort program if change found
  - issues: recompilation, debugger support



Compile-Time Defenses

Stack Protection

- Alternate method: at function entry, copy the return address in a safe, non-corruptible memory area
- upon function exit, check the return address on stack is the same as safe copy
  - abort the program in case changes are found

- format of stack remains unchanged.
- This approach is used in gcc extensions Stackshield and Return Address Defender (RAD).

Run-Time Defenses

Non Executable Address Space

aka Data Execution Prevention (DEP)

- Many BO attacks copy machine code into some buffer and transfer control to it
- Use virtual memory support to mark some regions of memory data-only, and non-executable (to avoid execution of attacker's code)
  - e.g. stack, heap, global data
  - need h/w support in Memory Management Unit (MMU)
- Issues: some applications need to put executable code in stack

Run-Time Defenses

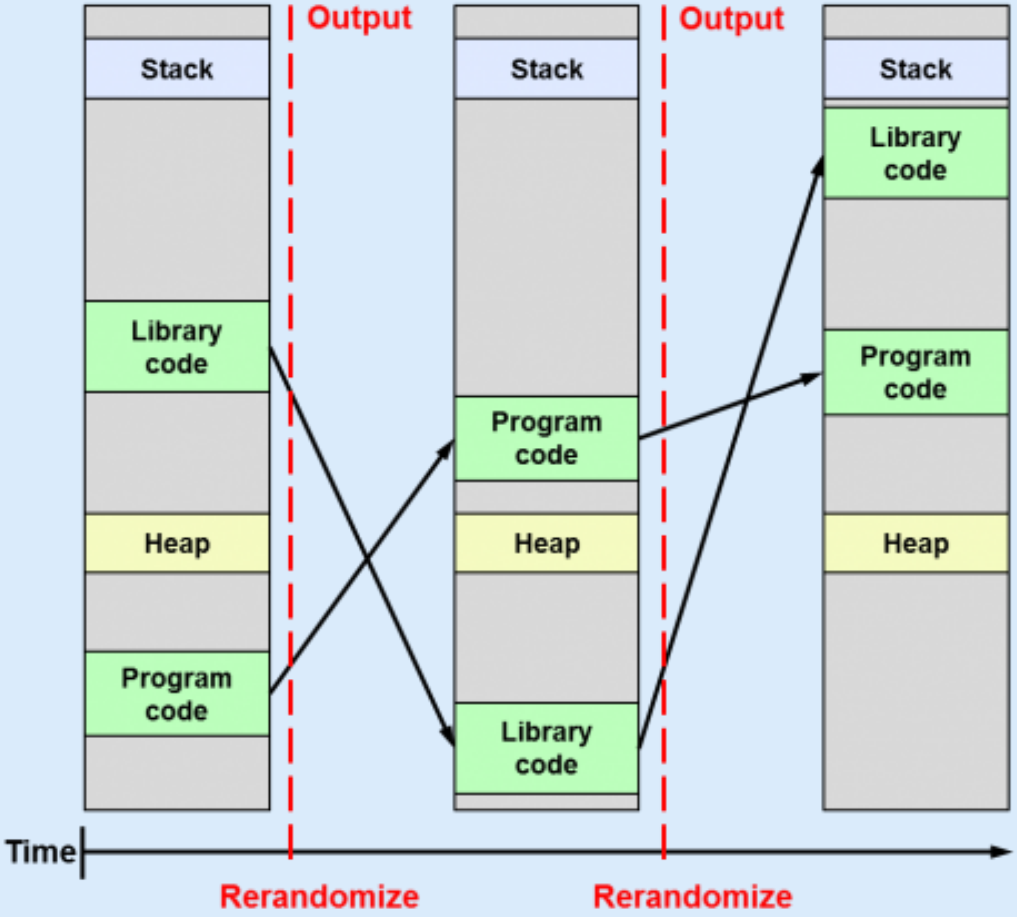
Address Space Layout Randomization (ASLR)

- Every time a process is created, randomize the location of key data structures in memory
  - stack, heap, global data: change addresses by 1 MB or so
  - using random shift for each process
  - modern systems have large address range, means wasting some has negligible impact

- Advantage: attacker can't predict the location of target buffer (containing the malicious code), so they can't transfer control to it

Run-Time Defenses

Address Space Layout Randomization (ASLR)



Run-Time Defenses

Guard Pages

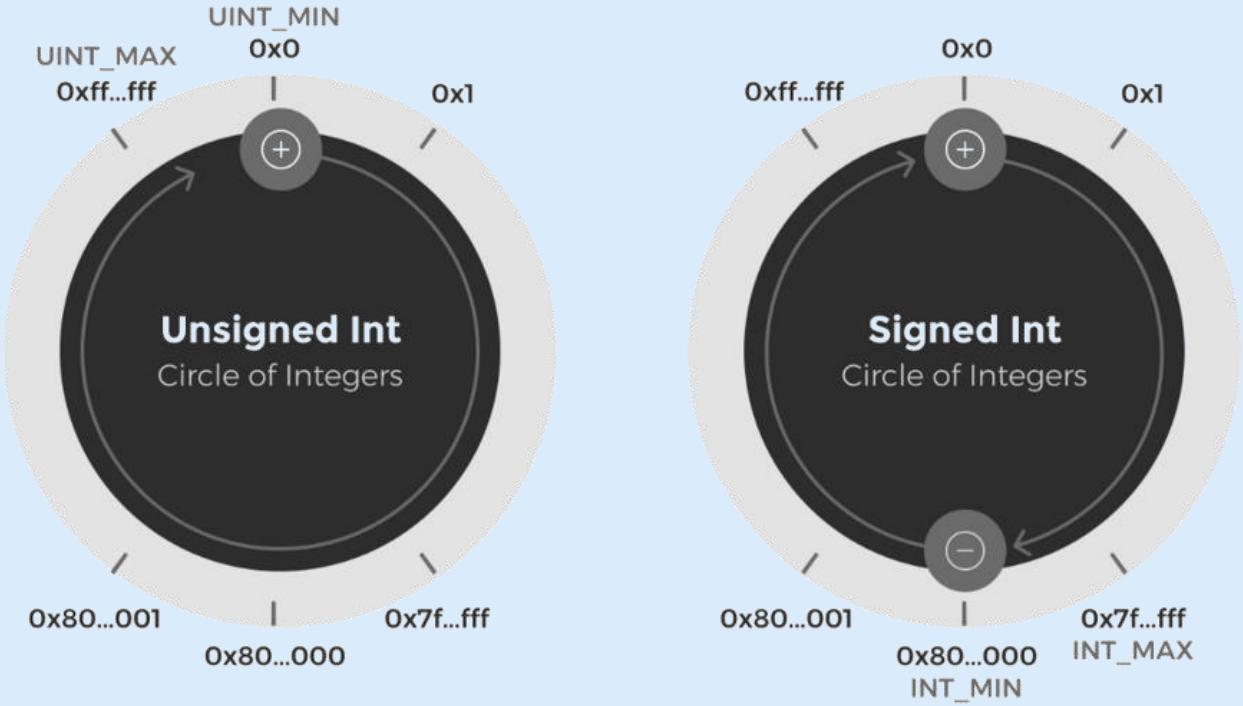
- Place guard pages between critical regions of memory (or between stack frames)
  - flagged in MMU (mem management unit) as illegal addresses
  - any access aborts process
- Can even place between stack frames and heap buffers
  - at execution time and space cost

Other Overflow Attacks

- There is a range of other attack variants
  - stack overflow variants
  - heap overflow
  - global data overflow
  - format string overflow
  - integer overflow
- more likely to be discovered in future
- some cannot be prevented except by coding to prevent originally

Integer Overflow

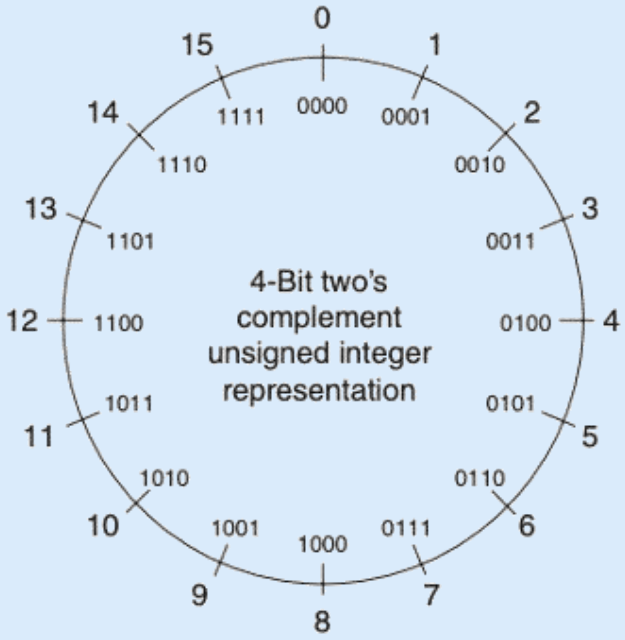
An arithmetic operation attempts to create a numeric value that is larger than can be represented within the available storage space.

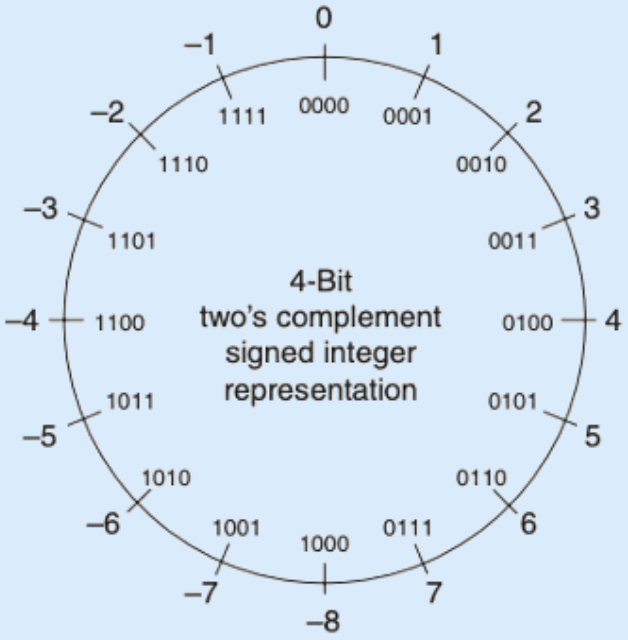


https://valid.network/post/integer-overflow-in-ethereum

Integer Overflow

Take 4-bit example for simplicity





Add two large unsigned numbers: 13+11
1101 + 1011 = 1 1000
which is 8 (after discarding the carry)

Add two large signed numbers: 5+7
0101 + 0111 = 1100 which is –4

https://medium.com/@ssserpilozdemir/what-is-integer-overflow-716db7d87c04

C/C++ Data Types

`char`	8 bits	[0; 255]
`short`	16	[-32,768; 32,767]
`unsigned short`	16	[0; 65,535]
`unsigned int`	32	[0; 4,294,967,295]
`int`	32	[-2,147,483,648; 2,147,483,647]
`long`	32	[-2,147,483,648; 2,147,483,647]

Note: These are typical sizes. Actual sizes may be different, based on platform (cpu architecture) and compiler.

Integer Overflow: Code Example

Overflow as a result of calculations

Test 1:

```
short x = 30000;
short y = 30000;
printf("%d\n", x+y);
```

Test 2:

```
short x = 30000;
short y = 30000;
short z = x + y;
printf("%d\n", z);
```

Assume `short` uses 16 bits
Will the two programs output the same?
What will they output?

Integer Overflow: Code Example

Overflow as a result of type casting

```
uint16_t x;                  // 16-bit variable
uint32_t y = 0x12345678; // 32-bit variable
x = (uint16_t) y + 1;
```

What value does x get?

the truncated value of y (lower 16 bits 0x5678) + 1

Where Does Integer Overflow Matter?

- Integer overflows, by themselves, do not allow direct overwriting of memory or hijacking execution flow control, but they indirectly lead to other security vulnerabilities (e.g. buffer overflow).

- How? Because overflowed integer could be used for following purposes
  - Allocating memory (by malloc, calloc etc.)
  - Calculating indexes into arrays
  - Checking whether a buffer overflow could occur

Integer Overflow Vulnerabilities

Example

```
const long MAX_LEN = 20000;
char buf[MAX_LEN];
short len = strlen(input);  // [1]
if (len < MAX_LEN)          // [2]
    strcpy(buf, input);
```

Can a buffer overflow attack occur?

If so, how long does the input needs to be?

Hints:

1. `strcpy` parameters are `(char *dest, char *src)`
2. `strlen()` returns a number of type `size_t` (unsigned number)

Integer Overflow Vulnerabilities

Example

```
const long MAX_LEN = 20000;
char buf[MAX_LEN];
short len = strlen(input);  // [1]
if (len < MAX_LEN)          // [2]
    strcpy(buf, input);
```

Range of short is [-32768, 32767]

When the length of input buffer exceeds the max 32767, the condition at [2] evaluates to true.

Note: when input length is between 32768 to 65535, the assignment [1] causes it to be treated as a negative number. When length exceeds 65535, assignment [1] results in truncation.

Another Example (from Phrack)

```
int copy_something(char *buf, int len) {
  int capacity = 800;
  char kbuf[capacity];
  if (len > capacity) {              // [1]
    return -1;
  }
  return memcpy(kbuf, buf, len); // [2]
}
```

What could go wrong?

Hints

- `memcpy` expects an unsigned integer as parameter.

Another Example (from Phrack)

```
int copy_something(char *buf, int len) {
  int capacity = 800;
  char kbuf[capacity];
  if (len > capacity) {              // [1]
    return -1;
  }
  return memcpy(kbuf, buf, len); // [2]
}
```

What could go wrong?

"By passing a negative value for `len` argument, it is possible to pass the check at [1], but then in the call to `memcpy` at [2], `len` will be interpreted as a huge unsigned value, causing memory to be overwritten well past the end of the buffer `kbuf`."

Integer overflow countermeasures

We can’t stop integer overflow, so we need additional code to either avoid it, or detect it

- In some architectures (e.g. x86), CPU itself checks for overflow and sets a flag. The program only needs to check for that flag.
- Otherwise manual code should be added for overflow detection, e.g. when calculating sum, if both numbers are positive and sum is negative then report an error. Similarly, if both numbers are negative and sum is positive then report the error.

String formatting in C

- Several available functions, like
  - print to screen (`printf`)
  - print to a file (`fprintf`)
  - print to a string buffer (`sprintf`)
- Syntax:

```
printf("format template", arg1, arg2, ...)
```

- Example

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

String formatting in C

Syntax: `printf("format template", arg1, arg2, ...)`

- The format template will contain one or more format specifiers, these control how `argN` will be encoded for display. e.g.

`%d`	Integer (base 10)
`%f`	Floating point
`%s`	String
`%x`	Hexadecimal integer
`%n`	Output nothing, but count characters printed so far and write that to `argN` pointer

- Optional field width specifier: `%4d` means output will be made at least four characters wide by prepending spaces (if needed). To prepend zeros instead, use `%04d`, `%08x` etc.

Malicious format strings: %x

What if args are missing?

```
printf("format template")
```

- If the format string contains one or more specifiers, `printf` will expect the args to be present on the stack.
- So it will read whatever is on the stack!

What could happen?

- Viewing the stack contents

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

- Output

```
16d2e5f8 00000000 e9a7ee80 e9cb2560 00000000
```

Malicious format strings: %s

- Typical use case

```
printf("str has the value %s", str);
// where %s is a string i.e. a char*
```

- What will the following do?

```
printf("str has the value %s");
```

- It interprets the top of the stack as a pointer (an address) and prints the string located in memory at that address.
- Of course, there might not be a string allocated at that address. `printf` simply prints whatever is in memory up to the next null terminator.
- It will either reveal (potentially sensitive) memory contents or crash the program if that address is illegal.

Malicious format strings: %n

- %n format specifier behaves differently. Instead of converting an arg to string, it counts the number of characters processed by printf so far, and stores that count to arg pointer.

```
int j;
printf("how long is this? %n", &j);
```

- Here it will give j the value 18
- Consequently,

```
printf("how long is this? %n")
```

will interpret the top bytes of the stack as an address X, and then write the number of characters outputted so far to that address!

Format string attacks

Leaking data from stack

```
int main(int argc, char** argv) {
   int pincode = 1234;
   printf(argv[1]);
}
```

How can an attacker learn the value of pincode ?

- Supplying %x%x%x as input will dump top 12 bytes of the stack

Format string attacks

- More serious outcomes
  - Reading from any address in process memory
  - Writing to any address in process memory
- Requires more investment of time and effort from the attacker
- See examples here:

https://web.ecs.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf

Vulnerable functions

Any function using a format string.

printf family:

```
printf, fprintf, sprintf, ...
vprintf, vfprintf, vsprintf, ...
```

Logging:

```
syslog, err, warn
```

Countermeasures

- Compilers can check if (static) format strings are malformed. e.g. in GCC compiler, the relevant flags are: -Wall, -Wformat, -Wno-format-extra-args, -Wformat-security, -Wformat-nonliteral, -Wformat=2.

"Most of these are only useful for detecting bad format strings that are known at compile-time. If the format string may come from the user or from a source external to the application, the application must validate the format string before using it. Care must also be taken if the application generates or selects format strings on the fly."

Wikipedia

Countermeasures

How do the malicious format strings get injected?

Whenever the format template is dynamic and externally sourced.

Vulnerable

```
int func(char *user) {
    printf(user);
}
```

Safe

```
int func(char *user) {
    printf("%s", user);
}
```

Because ‘user’ string could contain anything like %s, %x, %n. Attacker can craft an input string that could let it view or overwrite any location in memory.