Australian National University

Software Security

COMP2700 Cyber Security Foundations

Secure Software

- Software is secure if it can handle intentionally malformed input; the attacker picks (the probability distribution of) the inputs.
- Secure software: Protect the integrity of the runtime system.
- Secure software ≠ software with security features.
- Networking software is a popular target:
  - Intended to receive external input.
  - May construct instructions dynamically from input.
  - May involve low level manipulations of buffers.

Software Security

- There could be many underlying causes of security problems in software – these can be domain specific.
- But there are some common generic patterns of weakness; we'll cover a small selection of them.
- For a comprehensive list of common weakness in systems (software and/or hardware) see MITRE's Common Weakness Enumeration (CWE).
  - https://cwe.mitre.org

Outline

- A brief overview of the C language
- Integer overflow
- Canonicalisation
- Out-of-bound memory access
- Data as code

Australian
National
University

Overview of C

Why C?

- C is used in many implementations of operating systems, compilers, and system libraries.
- Relatively more efficient compared to other high-level languages.
- A major source of software bugs:
  - Mainly due to more flexible handling of pointers/references.
  - Lack of strong typing; easier for programmers to make mistakes.

Outline

We cover only some aspects of C relevant to this course:

- Pointers: arrays, memory allocation, pointer arithmetic
- Strings: representation, copying, concatenating.
- Format strings: in particular printf, a function to print output to standard output.

Helloworld.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   printf("Hello, World!\n");
   return 0;
}
```

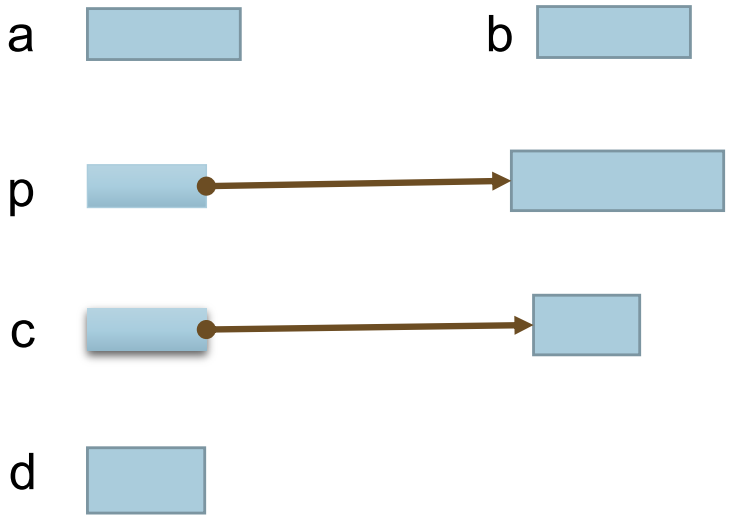
- Basic syntax similar to Java.
- Entry to program via “main” function:
  - argc: the number of arguments to the program
  - argv: an array of strings representing the arguments to the program
- printf: print a string to standard output

Some basic data types

- Integers:
  - int (16 bit or 32 bit): signed integers. For 32 bit int, value range is $-2^{31}-1$ to 2^{31} (one bit reserved for representing 'sign')
  - long (32 bit or 64 bit): signed integers. For 64 bit long, value range is $-2^{63}-1$ to 2^{63}.
- char (8 bit), representing characters.
- float: single precision floating points.
- Pointers: contain memory addresses. Syntax: add * to the type name.
  - E.g., int* denotes a type which is pointer to a memory location containing data of type int.

Variable Declarations

```
int a, b;

float * p;

char *c, d;
```



d is of type char, it is not a pointer

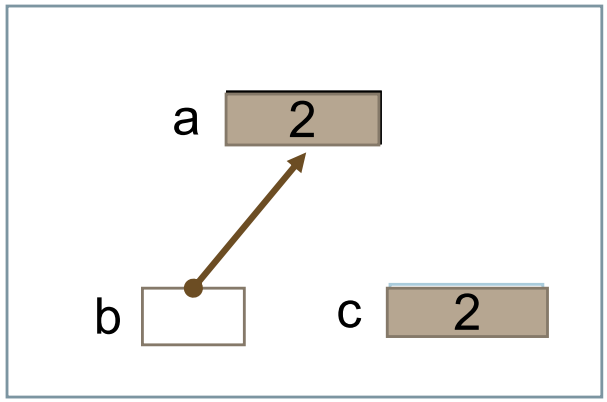
`char *c, d;` is the same as:

```
char* c;
char d;
```

The * and & operators

- We can get the memory location of a variable using the & operator.
- We can get the value of the memory location pointed to by a pointer using the * operator (dereferencing operator).

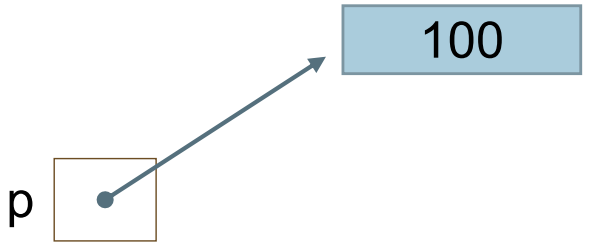
```
(1) int a;
(2) int *b;
(3) int c;

(4) a = 1;
(5) b = &a;
(6) ++a;
(7) c = *b;
```



Allocating memory

- The function malloc allocates a block of memory.
- It takes one argument specifying the size (in bytes) of the memory block to be allocated.
- If successful, pointer to the memory block is returned; otherwise the NULL value is returned.

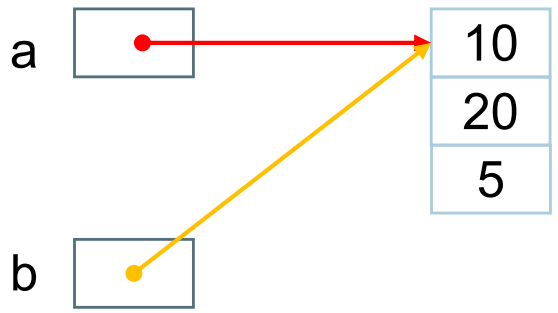
```
int *p;
p = (int*) malloc(sizeof(int));
*p = 100;
```



sizeof(int) returns the number of bytes for int type

Arrays

```
int a[3];
int* b;

a[0]=10;
a[1]=20;
a[2]=5;

b = a;
```

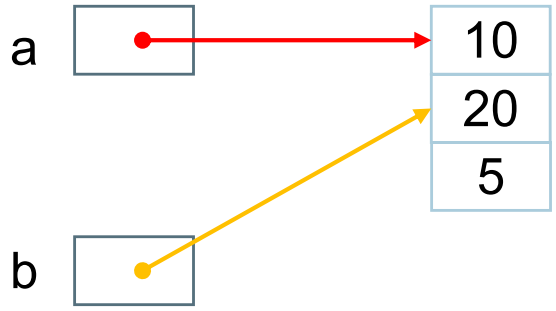


- The name of the array is a pointer.
- Note: there is no need to use & to get the address of the array.
- Array index starts at 0. So `a[3]` is outside the range of the array `a`.

Pointer arithmetic

You can add to or subtract from a pointer variable to access adjacent memory location.

```
int a[3];
int* b;

a[0]=10;
a[1]=20;
*(a+2)=5;

b = a + 1;
```

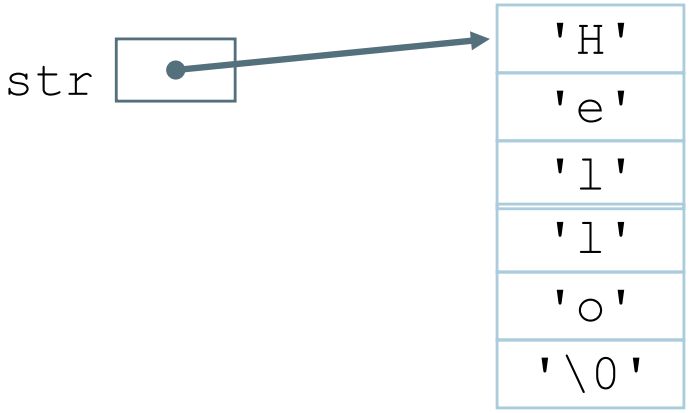


The notation `*(a+2)` is the same as `a[2]`.

Strings

- A string in C is just an array of char's.
- A string must end with a NULL (or '\0').
  - So an array of char of length n can hold only strings of length $n-1$. (The last character in the array is reserved for NULL.)
- Many string related functions rely on the NULL character to mark the end of a string; its absence could lead to unpredictable behavior and security bugs.

```
char str[6]= "Hello";
```



Operations on strings

- char* strcpy(char *dest, char *src): copying string src to dest.
- int strlen(char *str): return the length of the string str.
- char* strcat(char * dest, char *src): appends the string src to the end of the string dest.
- int strcmp(char *str1, char *str2): compare string str1 and str2.
  - Return value < 0: str1 ‘less than’ str2 (lexicographic ordering)
  - Return value > 0: str1 ‘more than’ str2
  - Return value = 0: str1 is equal to str2.

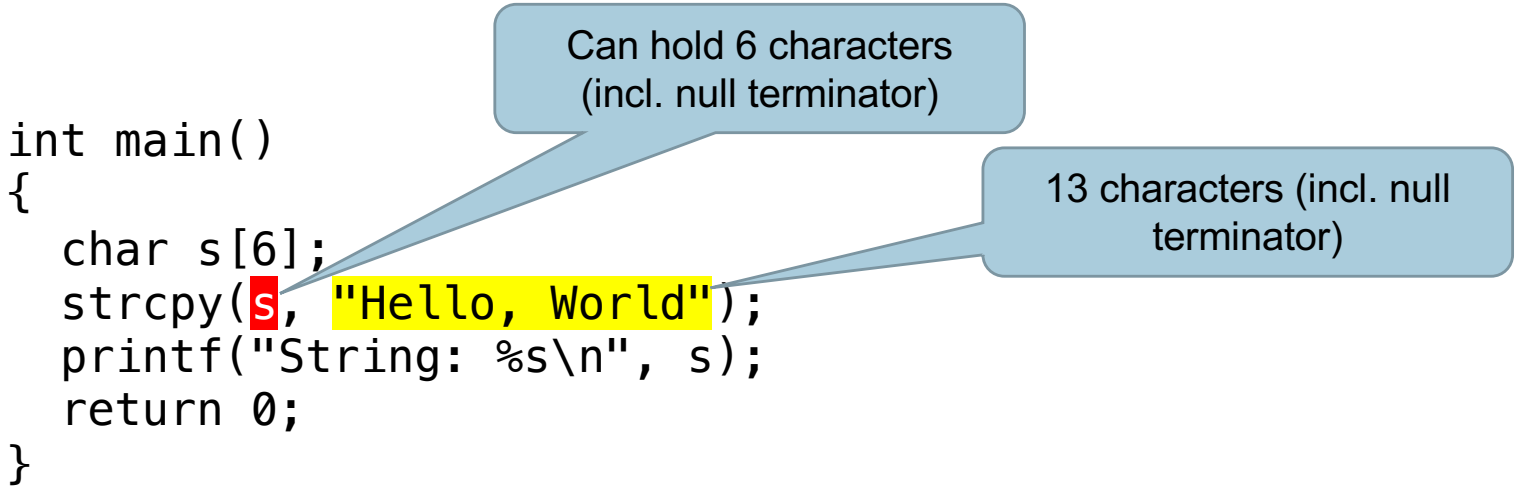
strcpy

This is how it is implemented:

```
char *strcpy(char *dest, const char *src)
{
  unsigned i;
  for (i=0; src[i] != '\0'; ++i)
    dest[i] = src[i];
  dest[i] = '\0';
  return dest;
}
```

- No checks on whether either or both arguments are NULL.
- No checks on the length of the destination string.

strcpy: what could go wrong?



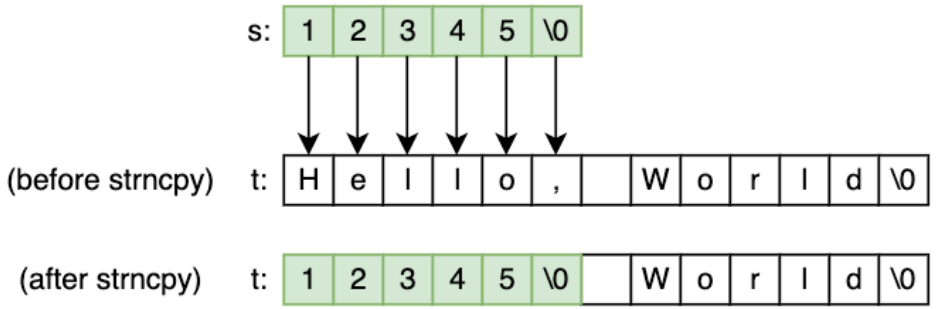
- Copying more characters than a destination buffer can handle could result in undefined behaviour.
- It is up to the programmer to ensure the destination buffer has enough space.

strncpy

- `char* strncpy(char *dest, char *src, int n)`: copy n characters from string src to string dest.
- With strncpy, there's no danger of over-copying.
- However, strncpy does not automatically add the NULL value ('\0') to dest if n is less than the length of string src.
  - This could make dest into a non-null terminated string.
- It is safer to always add a null terminator when using strncpy.

strncpy: correct usage

```
int main(int argc, char* argv[])
{
  char s[6] = "12345";
  char t[15] = "Hello, World";

  strncpy(t, s, 6);
  printf("String t: %s \n", t);

  return 0;
}
```



Output:

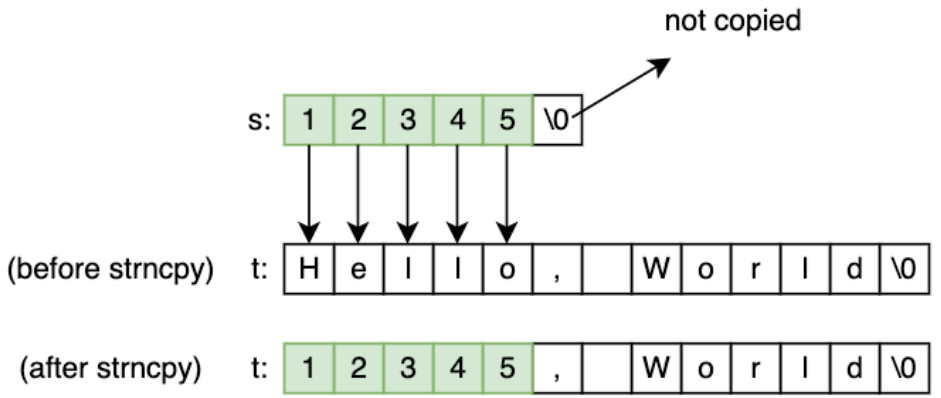
```
String t: 12345
```

strncpy: incorrect usage

```
int main()
{
  char s[6] = "12345";
  char t[15] = "Hello, World";

  strncpy(t, s, 5);
  printf("String t: %s \n", t);

  return 0;
}
```

Output:

```
String t: 12345, World
```



To be safe, if unsure, always add a null terminator explicitly.

```
strncpy(t, s, 5);
t[5] = '\0';
```

printf

- printf prints a format string to the standard output (screen).
- A format string is a string with special format specifiers (sequences of characters starting with `%').
- printf can take more than one argument. The first argument is always the format string; the rest consist of values to be substituted for the format specifiers.
- Format string is a main source of security problems.

printf examples

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

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

- printf("The first character in %s is %c", "abc", 'a');
  ```
  The first character in abc is a
  ```

- printf("The value of pi: %f", 3.14);
  ```
  The value of pi: 3.140000
  ```

Summary

- We focus only on a small part of the C language features.
- Most bugs related to C have to do with buffer operations, pointer manipulation and format strings; we'll see later how these are exploited.
- "The C Programming Language" book by Kernighan and Ritchie remains a very good and concise book to learn C.

Integer Overflow

Abstraction vs Implementation

- When writing code, programmers use elementary concepts like character, variable, array, integer, data & program, address (resource locator), atomic transaction, …
- These concepts have abstract meanings. For example, integers are an infinite set with operations ‘add’, ‘multiply’, ‘less or equal’, …
- Concrete implementations of these concepts may diverge from intuition – leading to vulnerabilities.
  - We'll see a concrete example involving integers

Programming with Integers

- In mathematics integers form an infinite set.
- In computer systems, an integer is represented as a sequence of bits of fixed length (precision).
  - So there is only a finite number of “integers”.
- Different "types" of integers: signed & unsigned integers, short & long (& long long) integers, ..
- In weakly typed languages (such as C), it is easy to mix up different types, resulting in truncation or misinterpretation of integers.

Computing with Integers

- Unsigned 8-bit integers

  $255 + 1 = 0$ $\qquad\qquad$ $16 * 17 = 16$

  $0 - 1 = 255$

- Signed 8-bit integers

  $127 + 1 = -128$

- In mathematics: $a + b \geq a$ for $b \geq 0$.
- Such obvious “facts” are no longer true in implementation.

What will happen here?

```
int i = 1;

while (i > 0)
{
   i = i * 2;
}
```

Two's Complement representation

- Signed integers are usually represented as 2's complement numbers.
- Most significant bit (sign bit) indicates the sign of the integer:
  - If sign bit is zero, the number is positive.
  - If sign bit is one, the number is negative.
- Positive numbers given in normal binary representation.
- In an N-bit signed integer, a sequence of N bits (where the left most bit is 1) is interpreted as the negative number obtained by subtracting the (unsigned) value of the bits from 2^N.

Two's Complement

- Consider 8 bit signed integers.
- To obtain the negative from a positive number, invert the bits, add 1.
- Example:

```
 5 = 0000 0101
-5 = 1111 1010 (invert)
             1 (add 1)
-------------- +
     1111 1011 (representation of -5)
```

Integer Overflow (CWE-190)

Integer overflow or wraparound (CWE-190) from cwe.mitre.org:

> The software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control.

Example: size overflow

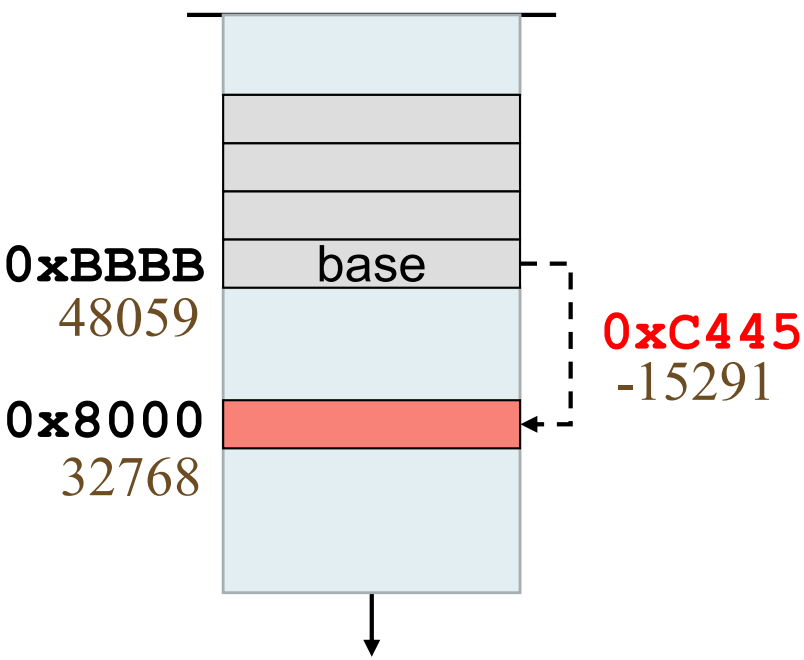
```
/*struct containing img data, 10kB each*/
img_t table_ptr;
int num_imgs;
...
num_imgs = get_num_imgs();
table_ptr = (img_t*)malloc(sizeof(img_t)*num_imgs);
...
```

Source: cwe.mitre.org

- The code intends to allocate a table of size num_imgs.
- However, if num_imgs grows large, the calculation (sizeof(img_t) * num_imgs) could overflow (wraparound), resulting in a small buffer.

Example: array indexing overflow

- You are given an array starting at memory location 0xBBBB (on a 16-bit machine)
- Array elements are single words.
- Which index do you write to so that memory location 0x8000 is overwritten?



0xBBBB + 0xC445 = 0x8000 mod 2^{16}

Example: integer overflow in kernel

- An example from Markus Kuhn (U. Cambridge).
- OS kernel system-call handler; checks string lengths to defend against buffer overruns.

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

len1 < sizeof(buf)

len2 = 0xffffffff

len2 + 1 = 2^{32}-1 + 1
= 0 mod 2^{32}

strncat will be executed

Integer Underflow (CWE-191)

Integer underflow (CWE-191) from cwe.mitre.org

The product subtracts one value from another, such that the result is less than the minimum allowable integer value, which produces a value that is not equal to the correct result. This can happen in signed and unsigned cases.

Example: subtracting from signed int

- Here `int` is a type for 32-bit signed integer.
- The value of `i` is already at the lowest negative value possible.
- Subtracting 1 from it results in a wrap around; its new value becomes 2147483647.

```
int main (void) {
   int i;
   i = -2147483648;
   i = i - 1;
   return 0;
}
```

Source: cwe.mitre.org

Example: casting signed to unsigned int

- Here size_t is 32-bit unsigned int.
- Although (a-b) gives -1 (as a signed int), interpreted as unsigned it, it is 4294967295.
- This program will allocate an extremely large buffer.

```
int main() {
  int a = 5, b = 6;
  size_t len = a - b;
  char buf[len];
  // blows up the stack
  …
}
```

Source: cwe.mitre.org

Summary

- Be aware of different types of integers and their underlying representations.
- Watch out for the cases where the (intermediate) results of arithmetic operations wraps around (overflow/underflow).
- Watch out for (implicit) type casting – especially in weakly typed languages.

Canonicalisation

Canonicalisation

- Canonicalization: the process that determines how various equivalent forms of a name are resolved to a single standard name.
- The single standard name is also known as the canonical name.
- In general, an issue whenever an object has different but equivalent representations.
- Canonicalization must be idempotent
  - The effect of applying the process once is the same as applying it twice.

Canonicalisation

Some common issues in canonicalisation:

- Relative path traversal (CWE-23)
- Case sensitivity (CWE-178)
- Handling of URL Encoding (CWE-177)

CWE-23: Relative Path Traversal

From CWE-23 (cwe.mitre.org)

- The software uses external input to construct a pathname that should be within a restricted directory, but it does not properly neutralize sequences such as ".." that can resolve to a location that is outside of that directory.
- This allows attackers to traverse the file system to access files or directories that are outside of the restricted directory.

Example: relative path traversal

```
int main()
{
    char cmd[128] = "cat /tmp/comp2700/";
    char input[32];
    printf("Input file name: ");
    fgets(input, 32, stdin);
    input[strlen(input)-1] = '\0';
    strcat(cmd,input);
    system(cmd);
    return 0;
}
```

The attacker can input "../../etc/passwd" to escape the "sandbox" /tmp/comp2700/

Examples: relative path traversal

A typical scenario: an application restricts the user to accessing file in a particular directory, does not filter the file name the user inputs. The user can use ".." to get around the restriction.

- "../somefile" : access traverse outside the current directory to access "somefile" in the parent directory.
- "../../etc/": if the current directory is two level down from root (/), this can be used to traverse to the /etc/ directory.

Example: relative path traversal

CVE-2021-41773: a path traversal vulnerability in Apache HTTP Server 2.4.49. Severity rating 7.5 (out of 10).

"An attacker could use a path traversal attack to map URLs to files outside the directories configured by Alias-like directives. …

If CGI scripts are also enabled for these aliased paths, this could allow for remote code execution."

CWE-178: Improper handling of case sensitivity

From cwe.mitre.org (CWE-178):

- The software does not properly account for differences in case sensitivity when accessing or determining the properties of a resource, leading to inconsistent results.
- Possible consequences:
  - Case-insensitive passwords, leading to reduced key space.
  - By-passing access controls using alternate names (with different mixture of cases)
- Example:
  - CVE-2021-45893 (case-insensitive passwords)
  - CVE-2022-22968 (access control by-pass)

CWE-177: Improper handling of URL encoding

- A Uniform Resource Locator (URL) is a string representation of a resource available on the internet. (RFC 1738)
- An URL is a sequences of letters, digits and some special characters.
- Arbitrary byte values can be encoded using a special character % followed by two hex digits.
  - For example: the letter '.' (dot) can be encoded as '%2e' (2e is the ASCII code for dot).
  - So https://www.google.com.au and https://www.google.com%2eau resolve to the same URL.

Example: URL handling vulnerabilities

- CVE-2021-23435: open redirect vulnerability.
  - By-passing a web server validation of a URL to redirect users to malicious websites.

- URL parsing can be quite messy, with different tools implementing slightly different interpretations. See:

  Moshe, et. Al. Exploiting URL parsers: the good, bad and inconsistent. 2022.

Summary

- An input may have different representations, and it is important to ensure access control decisions consider all representations (complete mediation).
- Best practice: always canonicalise before validating a representation.
  - The reverse (validate then canonicalise) is a common pattern of weakness (CWE-180).
- Some of the issues, e.g., relative path traversal, can be quite severe.
  - Among the 2022 CWE Top 25 Most Dangerous Software Weakness.

Data as Code

Scripting

- Scripting languages are used to construct commands (scripts) from predefined code fragments and user input.
- Script is then passed to another software component where it is executed.
- Scripting languages: SQL, bash (shell) scripts, Perl, PHP, Python, Tcl, Safe-Tcl, Javascript, …

Script injection

- The ***injection attack*** exploits improper validation of user input to a scripting language, causing parts of input to be interpreted as code.
  - Currently Injection is still among the top 10 security risks for web applications (OWASP Top 10).
- Attacks typically exploits certain features in a scripting language:
  - Symbols that terminate command parameters.
  - Symbols that terminate commands.
  - Dangerous commands, e.g. commands for executing the commands they receive as input (eval, exec, system, …).

SQL

- Structured Query Language (SQL) is a scripting language used for database management and operations.
- Strings in SQL commands are placed between single quotes.
- A typical attack against SQL is by "escaping" the quoting mechanism of SQL, and then injecting other SQL commands.
- SQL is commonly used in web applications that interact with databases, making this a popular attack target on web applications.

Example: SQL Injection

- Strings in SQL commands are placed between single quotes.
- Example query: select all records from the table 'client' in the database where the name field matches a user input ($name).

```
$sql = "SELECT * FROM client WHERE name= '$name' "
```

- In a normal usage, user would insert legal username like 'Bob' into query.

Example: SQL Injection

- Attacker enters as username:
  `$name = "Bob' OR 1=1 --"`

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

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

- Because (name = 'Bob' OR 1=1) evaluates to TRUE, the entire client database is selected; -- is a command telling SQL to ignore anything that would follow.

Shell Script Injection

- Unix shell script has some operators to compose commands, e.g.,:

`;`	`sequential composition`
`|`	`piping`
`<`	`input redirection`
`>`	`output redirection`
`$(command)`	`evaluating command`

(These are not the only possible commands; there are others.)

- User can inject shell commands into input to a script using one of the operators above.

Example: shell script injection

- Bash scripts are used in some web server to generate dynamic content, e.g., Apache's CGI.
- Example: a CGI (bash) script that sends file to $clientaddress:

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

- With the $clientaddress set to "to@me | rm -rf /", the server executes

```
cat file | mail to@me | rm -rf /
```

- After mailing the file to@me, all files the script has permission to delete will be deleted.

Example: shell script Injection

shell-inject.c

```
void main(void) {
  char cmd[256] = "/bin/cat ";
  char input[256];

  printf("File to display: ");
  fgets(input, 256, stdin);
  strcat(cmd, input);
  execl("/bin/bash", "bash", "-p", "-c", cmd, NULL);
}
```

The program ‘shell-inject.c’ asks for a filename and displays the file using the shell command /bin/cat.

- An attack works by feeding the program the input "somefile.txt;ls".
- The `bash` command treats the input "somefile.txt;ls" as two separate shell commands: "/bin/cat somefile.txt" and "ls".

Summary

- Many of the problems listed may look trivial in isolation, but finding them (in complex software) and assessing their security consequences are not obvious.

- General patterns in insecure software: familiar programming abstractions like variable, array, integer, data & code, address, or atomic transaction are being implemented in a way that breaks the abstraction.

- **Do not trust your input!** Always validate!

Out-of-bound Memory Access

Out-of-bounds memory access

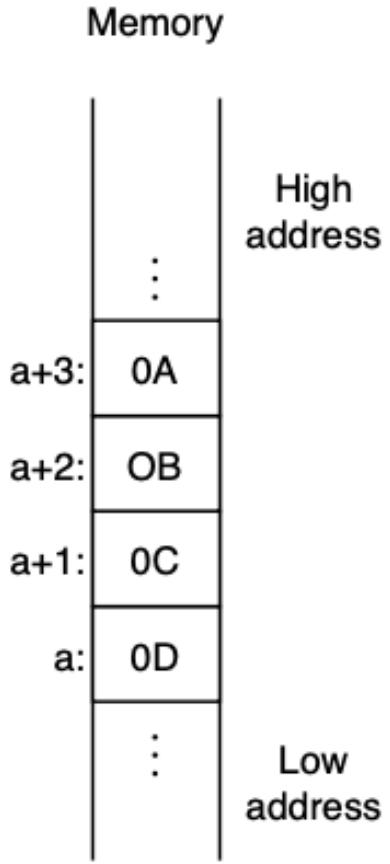
- CWE-787: Out-of-bounds Write
  - "The software writes data past the end, or before the beginning, of the intended buffer".
  - Number #1 in the 2022 CWE Top 25 Most Dangerous Software Weaknesses
- CWE-125: Out-of-bounds Read
  - ""The software reads data past the end, or before the beginning, of the intended buffer"
- Out-of-bounds writes can result in data corruption, crashes or code execution.
  - This is the kind of vulnerabilities that may give the attacker a complete system takeover.

Memory buffer

- A memory buffer refers to a concrete implementation of the concept of variables in programming languages.
  - Integers, strings, arrays, …
- Buffers allocated to variables may be adjacent to buffers allocated to hold meta-data for program execution (e.g., pointers to functions, return addresses, frame pointers, etc.)
- Depending on how buffers are placed in memory, an out-of-bound read/write of a variable may read from or write to the buffer of another variable, or critical meta-data.

Endianness

- An important concept in memory exploits is the endianness of the CPU architecture, i.e., the order of bytes of a data structure in memory.
- Most modern CPUs (e.g., Intel x86) use the little-endian format: the least significant byte at the smallest address.



Example: a 32-bit integer 0x0A0B0C0D at address *a*

Memory layout of a process

- In Linux, each process has its own virtual memory space.
- The most relevant regions (for this course) are the Stack and the Heap regions of the memory.
- Stack grows downward:
  - The top of the stack is at a lower memory address compared to the bottom of the stack.

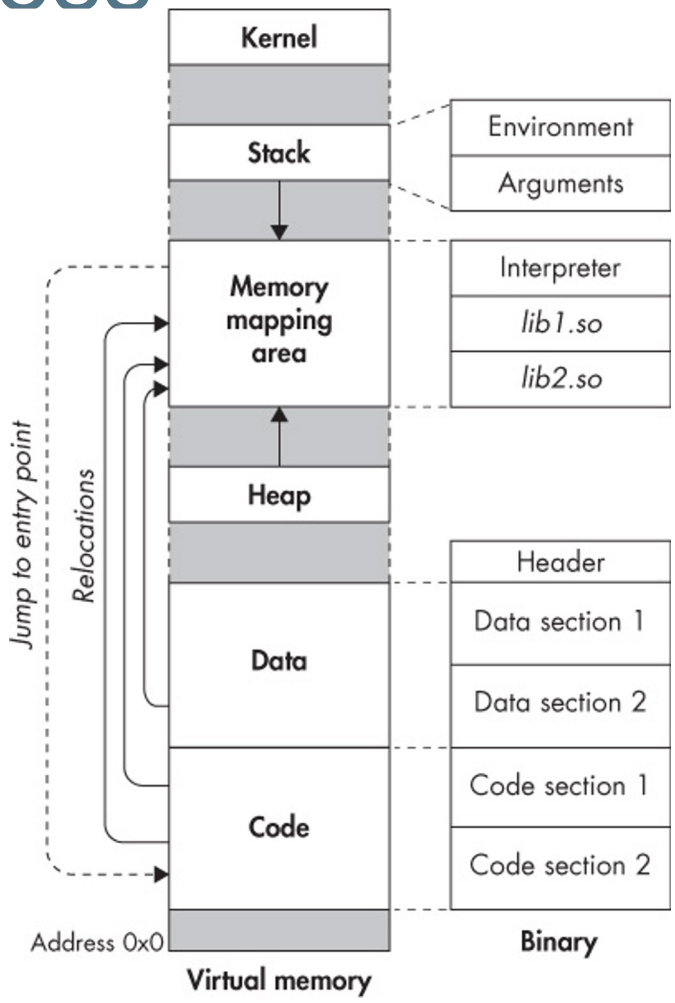
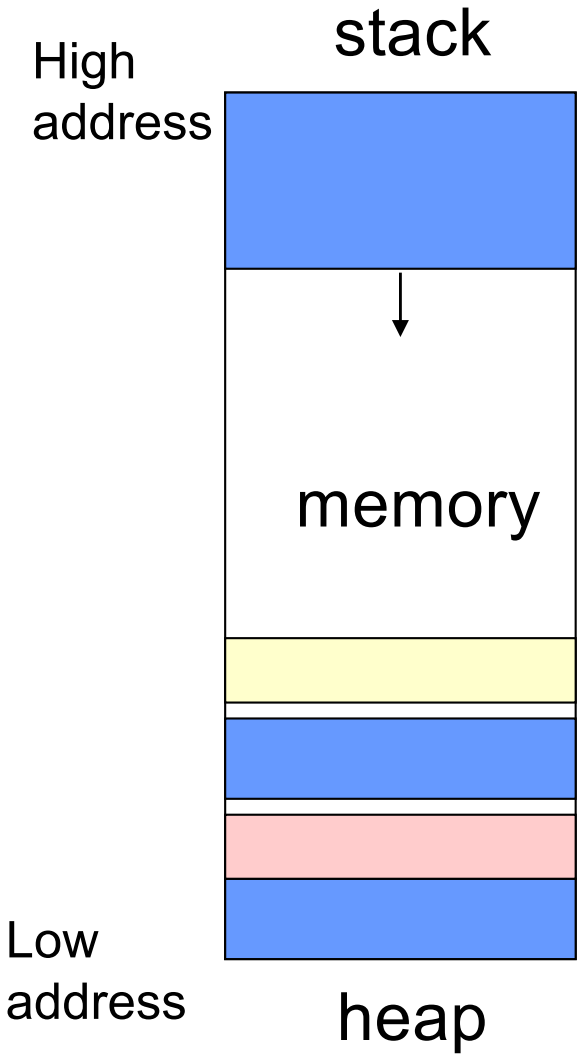


Image source: D. Andriesse. Practical Binary Analysis. 2019.

Memory Configuration

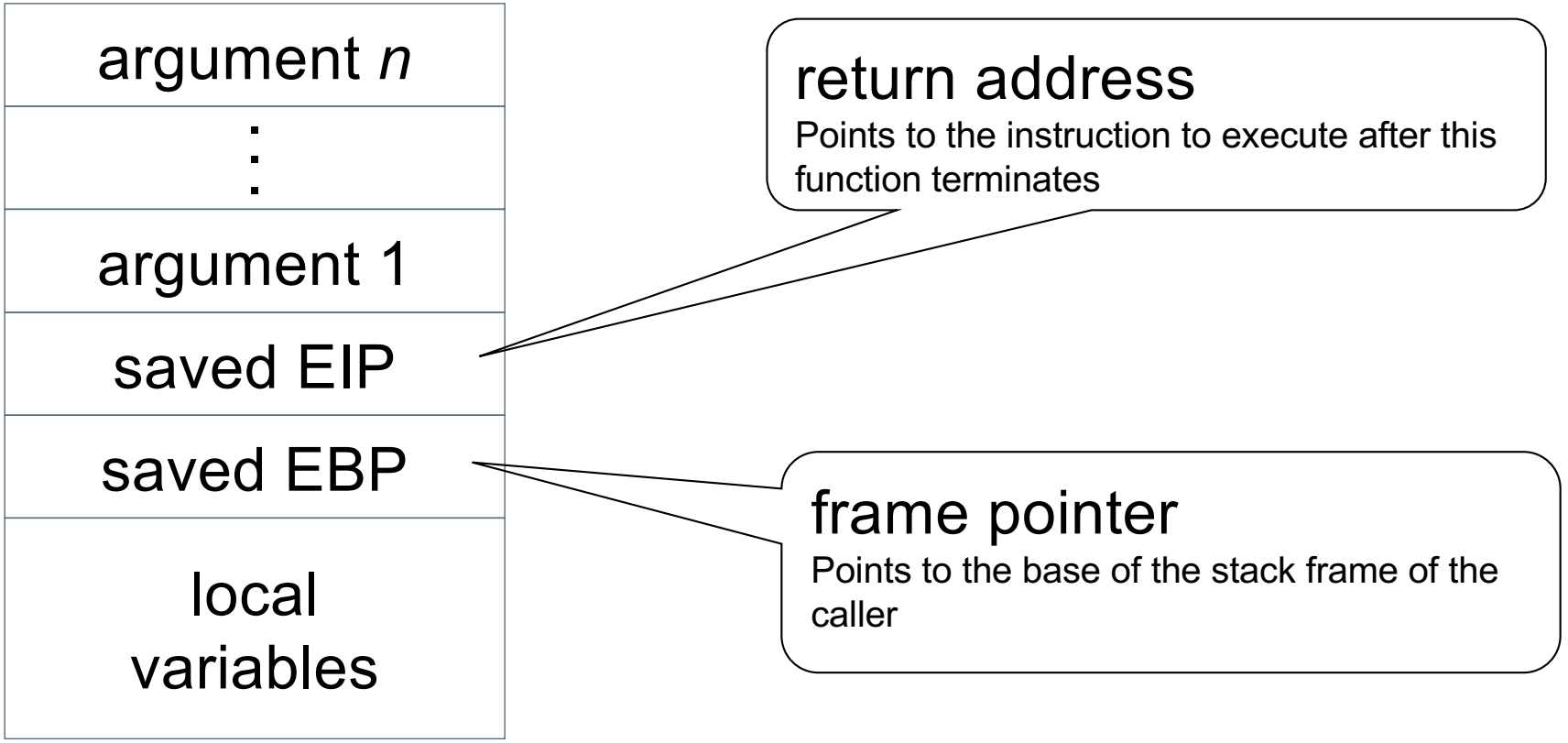
- **Stack**: contains stack frames, used to store return addresses, local variables and function arguments.
- **Stack grows downward**.
- **Heap**: contains dynamically allocated memory, .e.g., in C, memory allocated with malloc is placed in the heap region.



Stack Frames

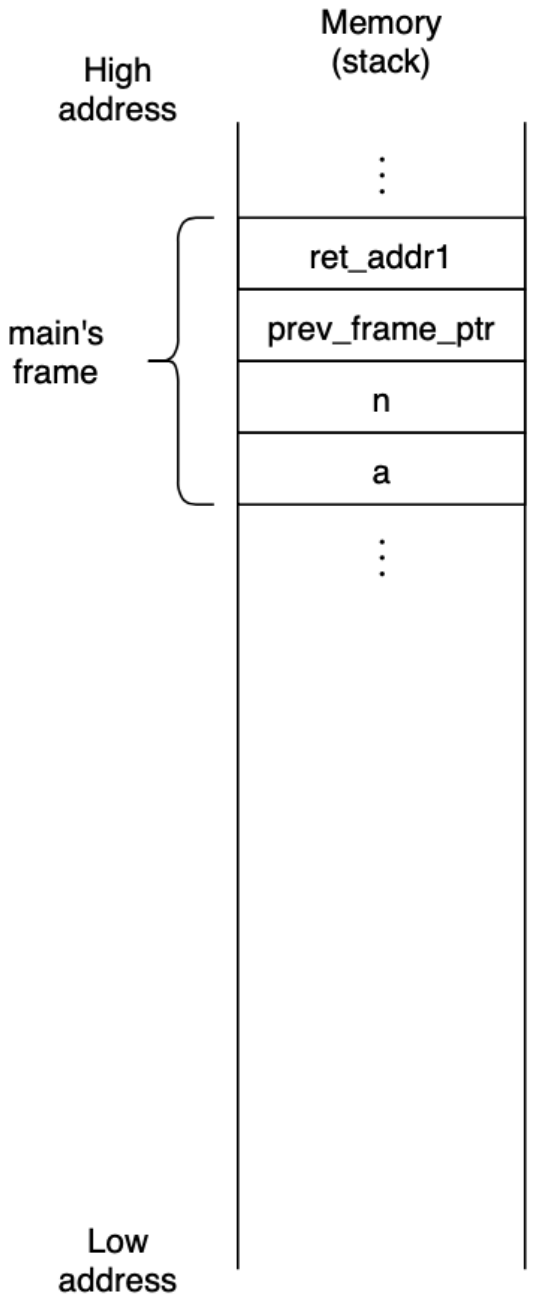
- In x86 architecture, for each function call, a memory configuration called stack frame is created and pushed to the stack region of a process.
- Each stack frame contains function arguments, meta-data (*return address* and *frame pointer*), and local variables of the function.
- When the call returns, the callee stack frame is freed and execution continues at the return address specified.

Stack Frame – Layout



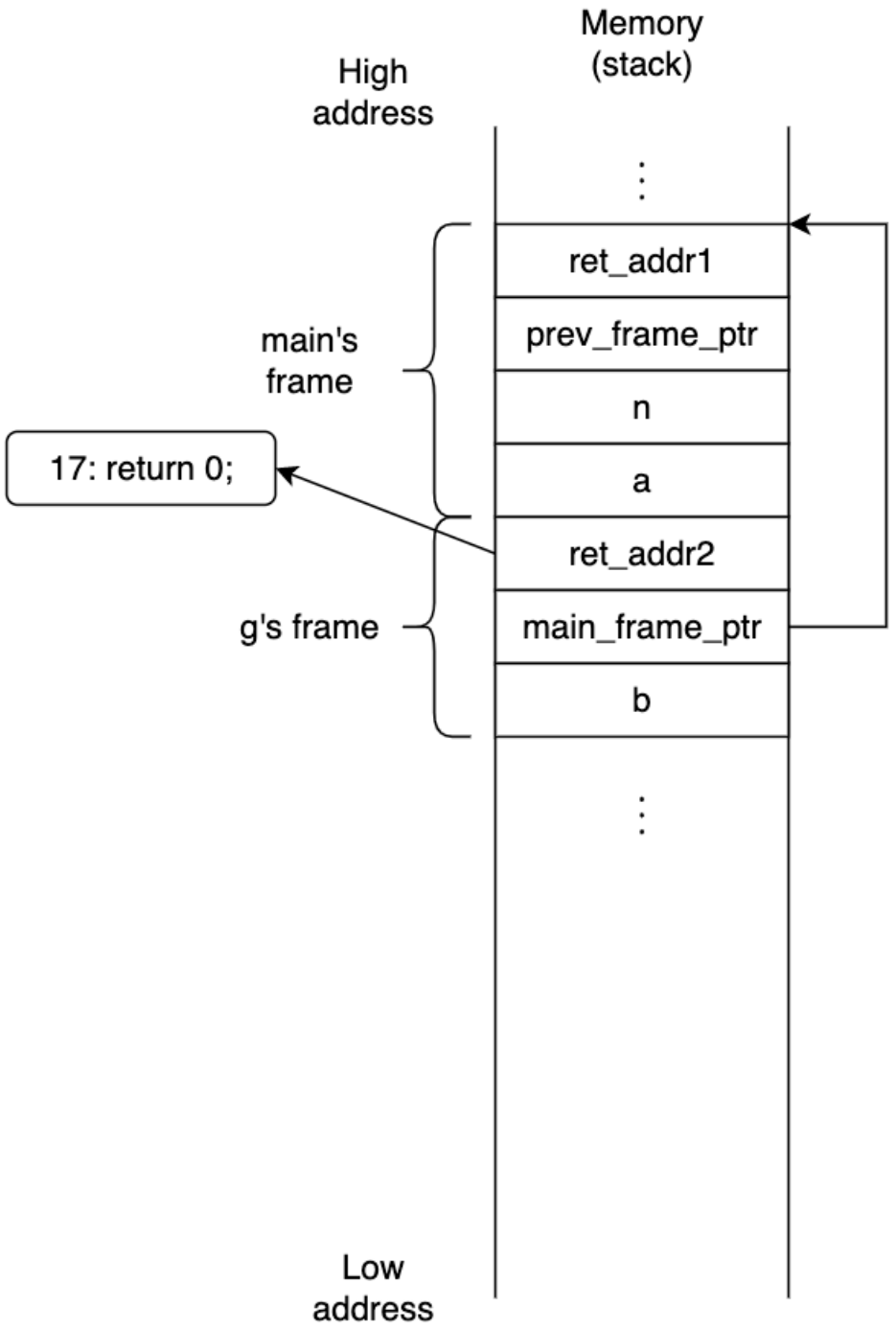
Example: call stacks

```
 4  int f(int x, int y) {
 5      int z;
 6      z = x + y;
 7      return z;
 8  }
 9  int g(int a) {
10      int b;
11      b = f(a,a);
12      return b;
13  }
14  int main(void) {
15      int n = 2;
16      g(n);
17      return 0;
18  }
```



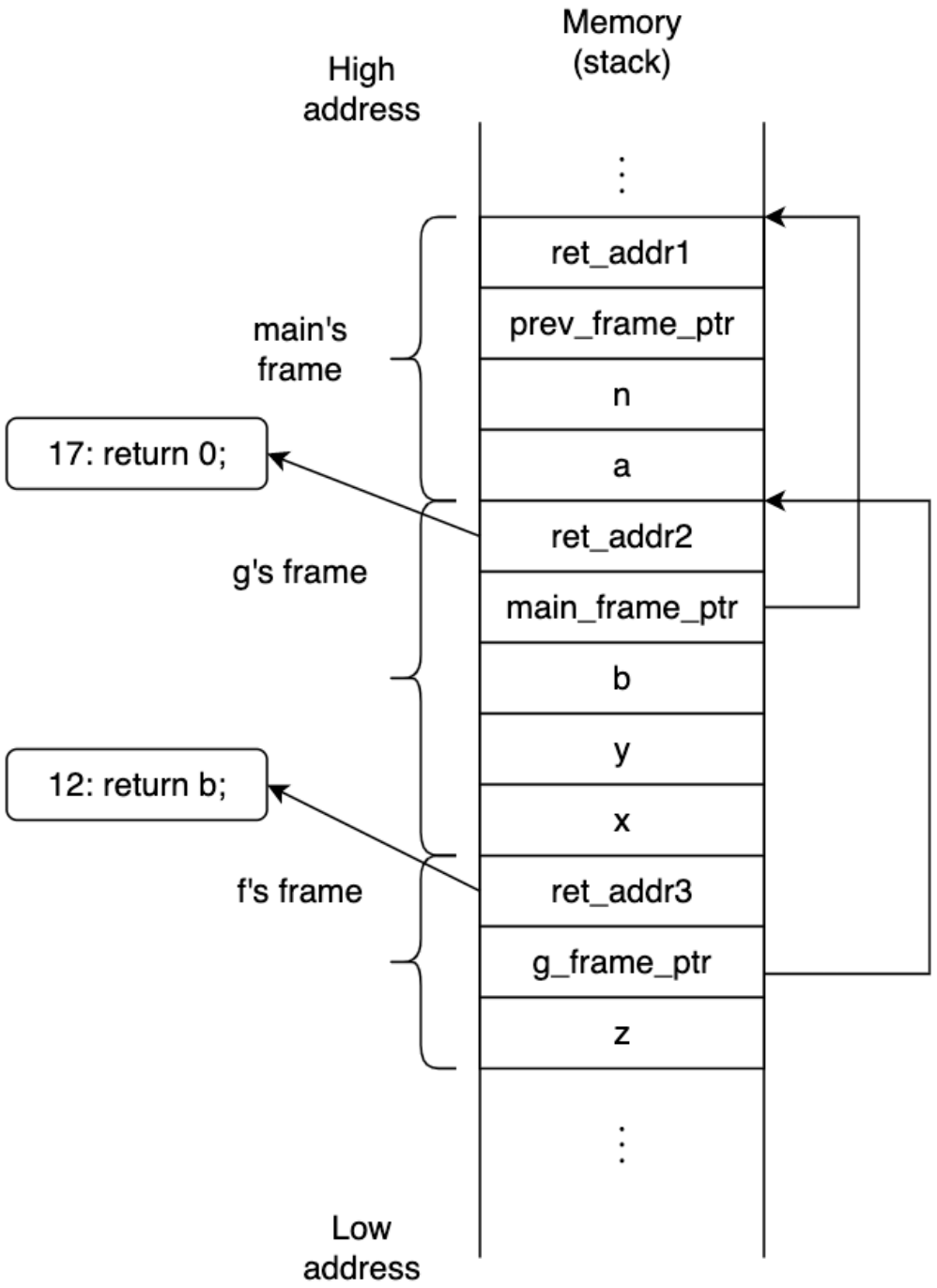
Example: call stacks

```
 4  int f(int x, int y) {
 5      int z;
 6      z = x + y;
 7      return z;
 8  }
 9  int g(int a) {
10      int b;
11      b = f(a,a);
12      return b;
13  }
14  int main(void) {
15      int n = 2;
16      g(n);
17      return 0;
18  }
```



Example: call stacks

```
 4  int f(int x, int y) {
 5      int z;
 6      z = x + y;
 7      return z;
 8  }
 9  int g(int a) {
10      int b;
11      b = f(a,a);
12      return b;
13  }
14  int main(void) {
15      int n = 2;
16      g(n);
17      return 0;
18  }
```



Sources of out-of-bound access

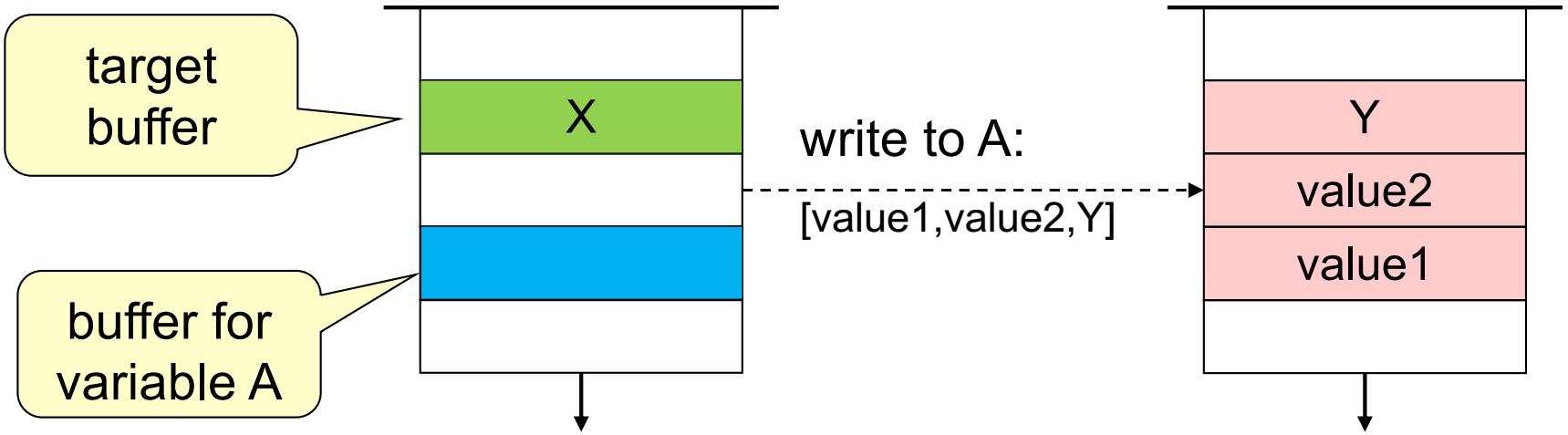
- Conditions that trigger out-of-bound access vulnerabilities are typically specific to the program logic.
- But some library functions are more prone to misuse than others, to create these vulnerabilities.
  - String copying operations: strcpy, strcat, …
  - Format strings: printf, sprintf, fprintf, …
- Generally, functions that process elements of arrays may be a source of vulnerabilities.
- We will examine exploit techniques that target the stack area of a process.

Stack-based buffer overflow

- The most well-known class of out-of-bound write vulnerabilities is the stack-based buffer overflow.
- This exploits weakness in software to overwrite data in the stack frame of a function.
- A popular target is the return address stored in the stack.
  - This allows the attacker to execute arbitrary code.
  - See Aleph One's "Smashing the stack for fun and profit.".
- But one could also target local variables or arguments of a function, to affect limited changes to control flow.

Stack-based buffer overflow

- Find a vulnerability that allows an out-of-bound write to fill a local buffer.
- Overwrite the target address with values of your choice.
  - The actual values will be specific to the desired effect.



Example: Stack-based overflow

buf_overflow.c

```
int main(int argc, char *argv[]) {
  char a = 'a';
  char buffer[16];
  if(argc < 2){
    printf("Usage: %s <string>\n", argv[0]);
    exit(0);
  }
  strcpy(buffer, argv[1]);
  printf("Value of a: %c\n", a);
  return 0;
}
```

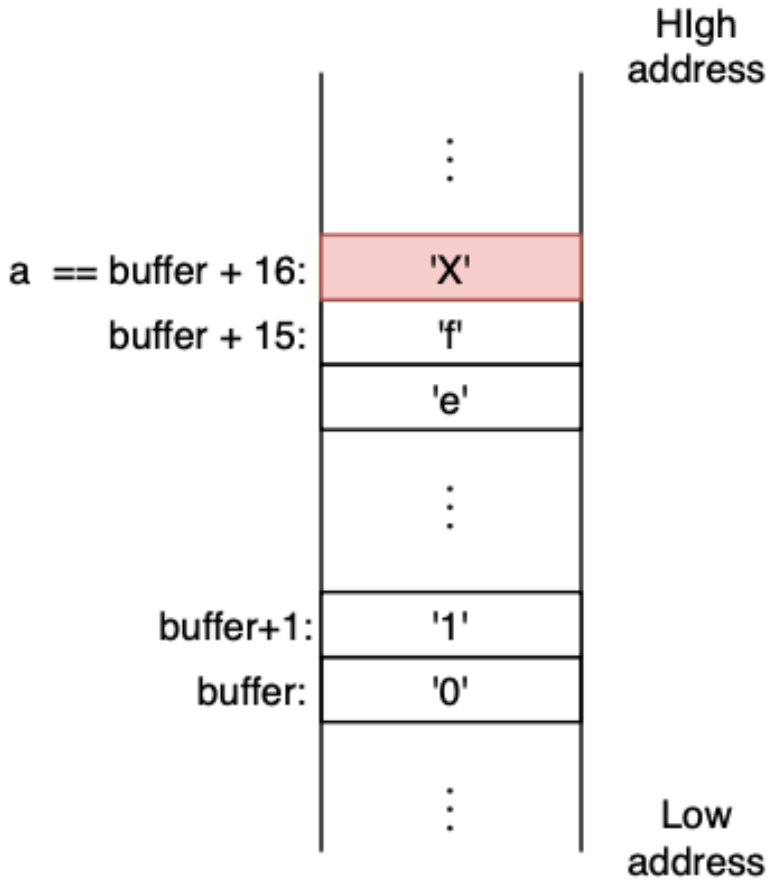
Example: Stack-based overflow

- Compile the program:

```
gcc buf_overflow.c –o buf_overflow –fno-stack-protector –m32
```

- Try run the program with input: **`0123456789abcdefX`**
- What is the value of **`a`** printed at the end of the program?

Example: Stack-based overflow



Overflowing `buffer` will overwrite the memory location for the variable `a`.

Another example

```
int check_authentication(char *password)
{
  int auth_flag = 0;
  char buffer[16];
  strcpy(buffer, password);

  if(strcmp(buffer, "alice") == 0)
     auth_flag = 1;
  if(strcmp(buffer, "bob") == 0)
     auth_flag = 1;

  return auth_flag;
}
```

From Jon Erickson's "Hacking: the art of exploitation":

Another example

```
int check_authentication(char *password)
{
  int auth_flag = 0;
  char buffer[16];
  strcpy(buffer, password);

  if(strcmp(buffer, "alice") == 0)
     auth_flag = 1;
  if(strcmp(buffer, "bob") == 0)
     auth_flag = 1;

  return auth_flag;
}
```

No check on the length of password

From Jon Erickson's "Hacking: the art of exploitation":

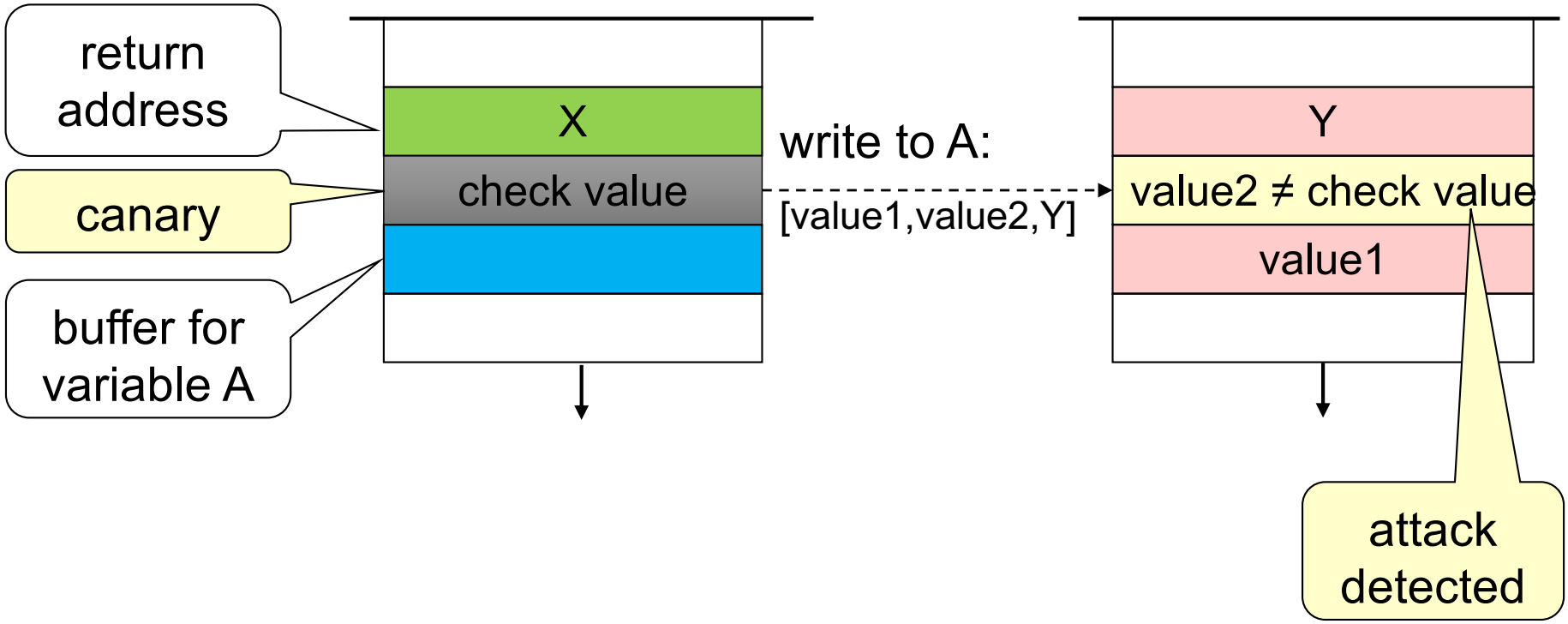
Another Example

- If authentication is successful, the function returns a non-zero value. Otherwise, it returns 0.
- The valid passwords are hardcoded (‘alice’ and ‘bob’).
- How do you get authenticated without knowing the passwords?
  - Input a password that is long enough to overflow `buffer` and overwrites `auth_flag`.

Stack canaries

- Aimed at detecting attempts at overwriting the return address.
- Place a check value (‘canary’) in the memory location just below the return address.
- Before returning from the function, check that the canary has not been changed.
- Example: gcc's Stackguard. Uses random canaries.
- Source code has to be recompiled to insert placing and checking of the canary.

Canaries


return address
canary
buffer for variable A
X
check value
write to A:
[value1,value2,Y]
Y
value2 ≠ check value
value1
attack detected

Format String Vulnerability

- A format string is a character string with special ‘format specifiers’.
  - Example: in printf function in C, format specifiers are those prefixed with %, e.g., %d, %x, etc.
- Format specifiers are place holders.
  - They will be replaced by other values when the format string is printed.
- Format specifiers are essentially instructions.
  - Attack works by injecting format specifiers into format strings.
- Format string vulnerability can be used for both out-of-bound read and write.

Format strings: printf

- The printf function takes variable-length arguments:
  - The number of arguments depends on the number of format specifiers in the format string.

```
int x=1, y=2;
printf("Value of x: %d, value of y: %d", x, y);
```

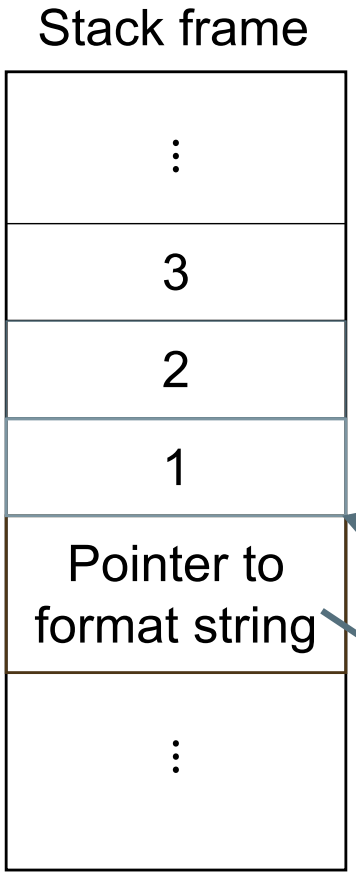
- C compiler does not check that the number of format specifiers matches the number of arguments.

```
int x=1, y=2, z=3;
printf("Value of x: %d, value of y: %d, value of z: %d",
        x, y);
```

This program is accepted by the compiler.

printf's arguments

```
printf("First: %d, second: %d, third: %d", 1, 2, 3);
```



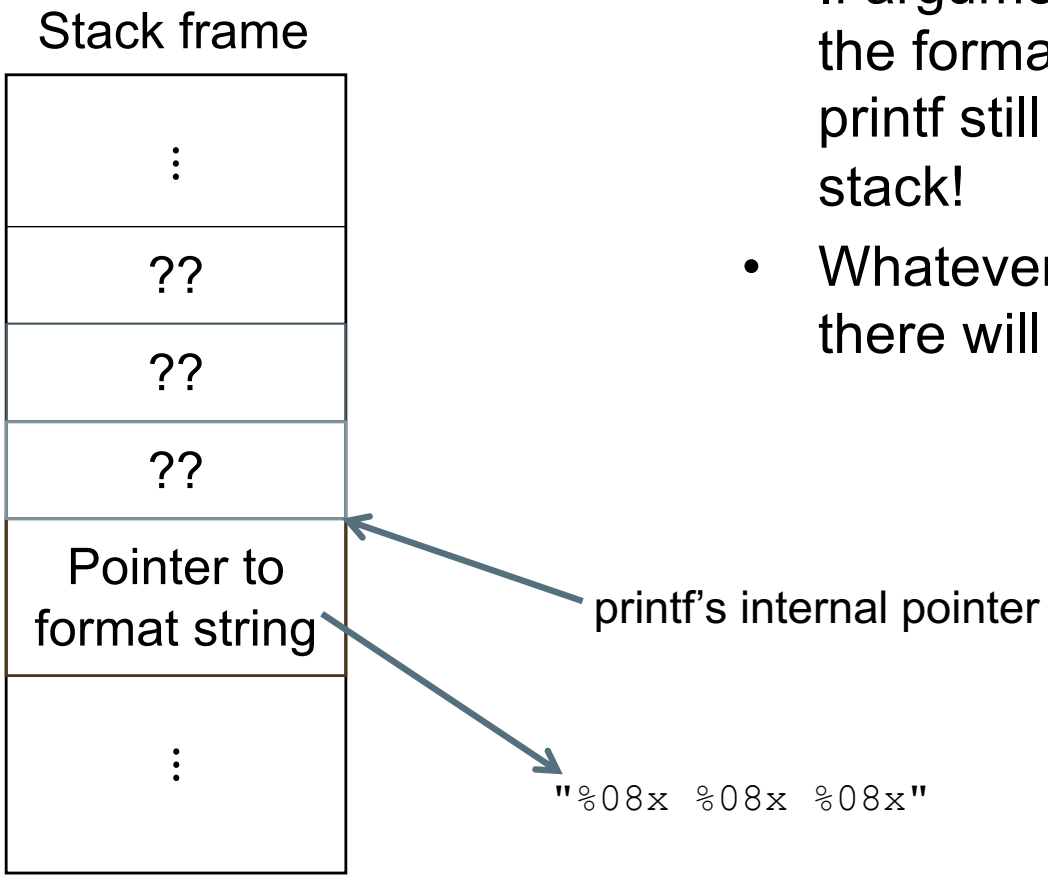
- Internally, printf keeps a pointer that points to list of its arguments in the stack.
- By convention, the arguments are whatever values in the stack, above the format string.
- The caller of printf is responsible for pushing those arguments to the stack.

printf's internal pointer

`"First: %d, second: %d, third: %d"`

printf: out-of-bound read

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



- If arguments in printf do not match the format specifiers, at runtime, printf still retrieves values from the stack!
- Whatever values that happen to be there will be printed.

printf: out-of-bound write

- A printf format string vulnerability can also be used for out-of-bound write, using the format specifier %n.
  - %n is used to write the number of characters printed so far to a designated variable.

```
int k;
printf("aaaaa%n", &k);
```

Print five characters ('aaaaa') and write the number 5 to variable k.

```
int k;
printf("aaaaa%n");
```

Print five characters ('aaaaa') and write the number 5 to an undetermined memory location (depending on what's on the stack).

- This may allow an attacker to write to an arbitrary memory location.

Prevention

- Format string vulnerabilities can be more powerful than stack-based overflow through string-copying operations.
- Fortunately, it is easy to mitigate.
- Prudent practice: never let the user controls the format string.
  - e.g., statement like printf(str), where str is a variable controlled by the user.
  - Modern C compilers have already built-in detection of potential printf format string vulnerability.
- Safe pattern: replace printf(str) with printf("%s",str).

Summary

- Out-of-bound access is one of the most severe vulnerabilities.
- There are many possible causes – we looked only at stack-based overflow and format strings.
- Stack-based exploits are mostly mitigated in modern systems.
- But other exploits against heap (not covered in this course) are possible, and are often more powerful.