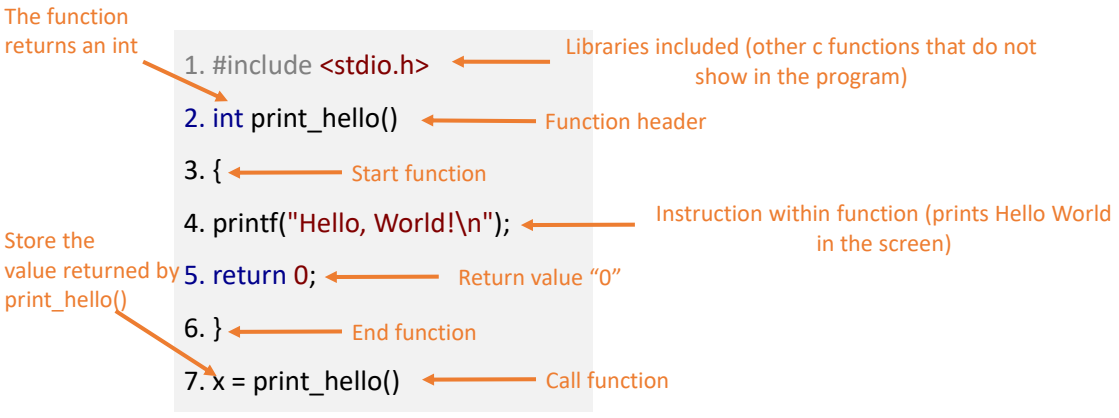
Midterm

C programming – Technical Preliminary
Reminder for a security lecture

C language 101: concepts for the lecture
(not a programming course)

Low-level general-purpose programming language



These are basic concepts in C to follow the lecture. Please check books or online tutorials to gain familiarity with these concepts.

Suggestions for online tutorials:
- https://www.guru99.com/c-programming-tutorial.html
- https://www.learn-c.org/

Note that you do not need to learn to program in C, but you do need to understand concepts such as function calls, pointers, variable, types, etc.

Note: even though we won't ask you to write a lot of C, at all, we will ask you to know how to read C. And writing C is a good way to learn to read C. If you are struggling I think a good advice is to write a little bit of C.

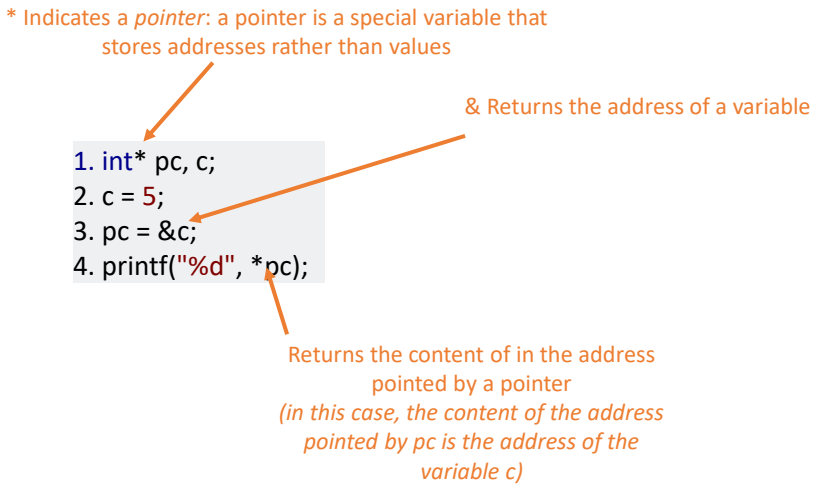
C language 101: concepts for the lecture
(not a programming course)

```
1. int addNumbers(int a, int b)
2. {
3.   int result;
4.   result = a+b;
5.   return result; // return statement
6.}
```

Function receives 2 integers (a, b) and returns an integer

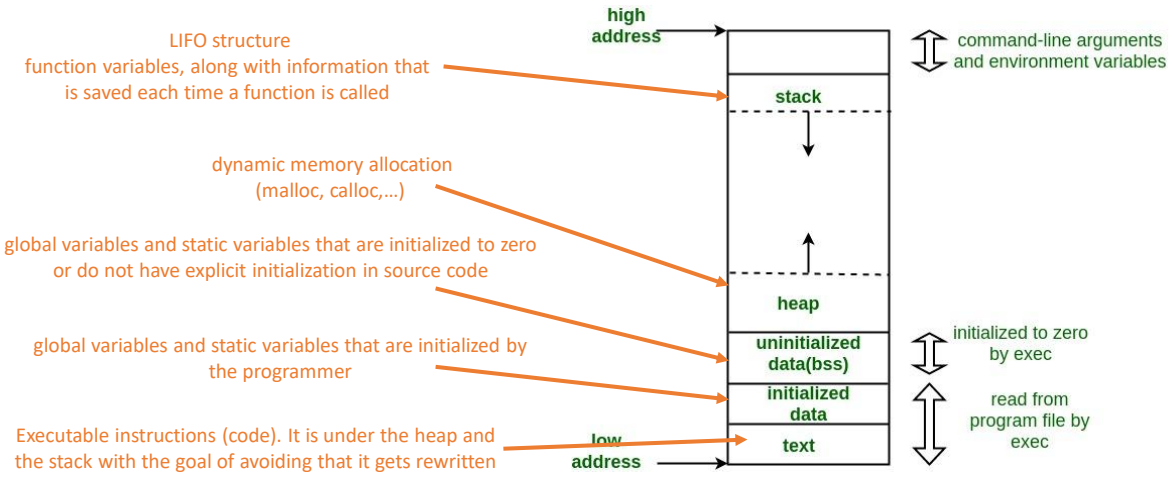
A local variable, only exists inside the function

C language 101: concepts for the lecture
(not a programming course)



C language 101: concepts for the lecture
(not a programming course)

Layout of a C program



In order to understand software vulnerabilities, it is necessary to know how the different variables and objects in a C program are laid out in memory. The figure illustrates where global, local, and dynamically allocated variables are placed in memory.

In particular it is important to understand heap and stack. This link can help: https://www.gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html but there are many other resources online that can help improving your understanding of memory allocation.

Exercise 1: The Geography of Variables

```
int counter = 0; // (A)
void process_data(int size) { // (B: 'size')
  char *buffer = malloc(size); // (C: 'buffer' pointer itself)
                               // (D: what 'buffer' points to)
  char *static_str = "Fixed"; // (E: the string literal "Fixed")
  free(buffer);
  g(buffer);
}
```

1. Map the letters (A-E) to their memory segments: Stack, Heap, Data/BSS, or Text (Read-only data).

2. At the end of process_data, we call g
   Can we still access the address stored in variable buffer within function g?
   Can we access the data at D within function g?

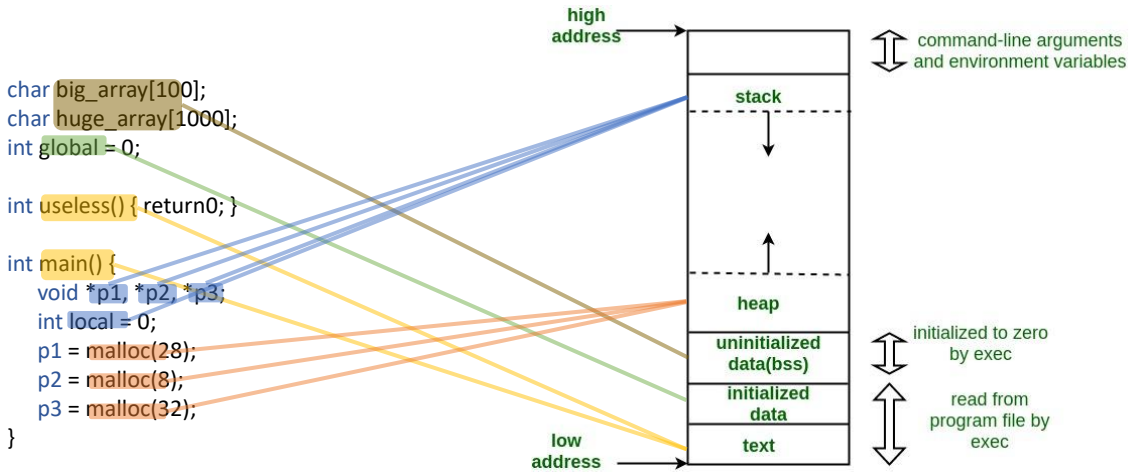
3. How many allocations happened on the **Heap**?

A (Data/BSS): Global variable.
B (Stack): Function argument.
C (Stack): The pointer variable *itself* lives on the stack frame.
D (Heap): The memory block allocated by malloc
E (Text/RODData): String literals usually live in read-only memory.

Q2: You can access C (the pointer exists until the function returns), but accessing D is a Use-After-Free (undefined behavior). The data might or might not be there, hence why it is soooo dangerous.
Q3: Exactly one.

C language 101: concepts for the lecture

Layout of a C program



This figure illustrates the concepts of the previous slide and how different variables map to memory.

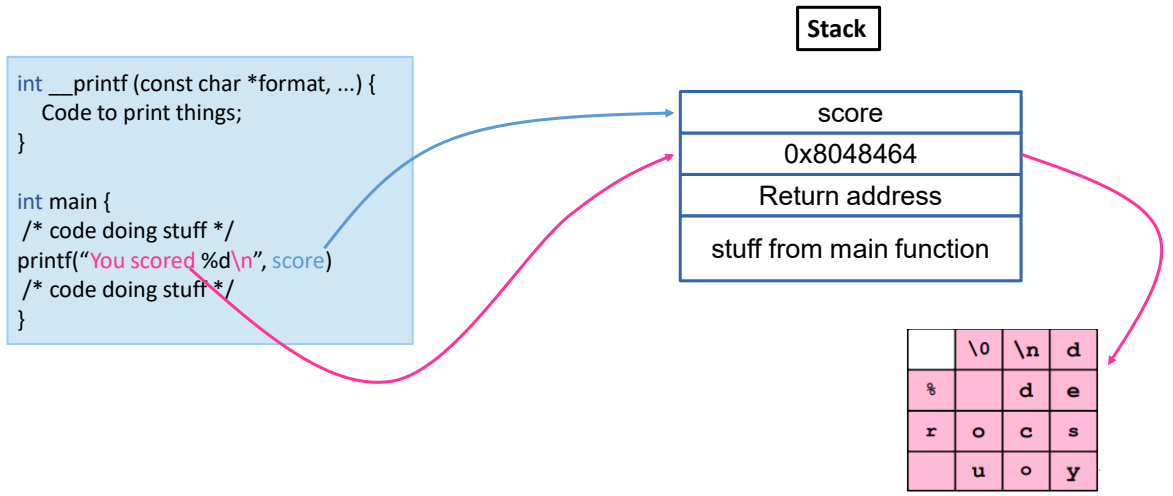
When a function is called, it reserves a “stack frame”: space in the stack for its variables.

Stack frames are reserved “on top” of each other according to how the stack grows. Whether the stack grows upwards or downwards depends on the architecture, but within an architecture it is consistent. In most architectures, the stack grows downwards.

Within the stack frame, most architectures allocate space for the variables as they come (see lecture 7.2); but in some the order can be random.

C language 101: concepts for the lecture

Calling a function



Explanation about how C programs store the addresses and values of variables in the stack and in memory.

`printf` is a function that takes one or more parameters:

- The first parameter (written above the return address on the stack) is the address of a string to be printed in the screen. This string may contain format specifiers which indicate that subsequent parameters will be variables whose values will be plugged in the string when it is printed.
- If the string has format specifiers, these variables are given as parameters to the `printf` function, and as such are also on the stack.

Exercise 2 - Anatomy of Memory - Code Is Data

```c
typedef void (*func_t)();
void secret_function() { printf("Win!\n"); }

void trigger() {
    // 1. Allocate space for a function pointer on the HEAP
    func_t *heap_hook = malloc(sizeof(func_t));

    // 2. Store the address of the code into that heap memory
    *heap_hook = secret_function;

    // <-- SNAPSHOT TAKEN HERE
    free(heap_hook);
}
int main() {
    trigger();
    return 0;   // Line 16
}
```

Q1 Memory Layout Analysis: At the snapshot, where does the `heap_hook` live (stack/heap…), where does it points to, what kind of **value** is stored inside the location it points to? (Is it a number? An instruction? An address?)

Q1: The heap contains the **Address of the secret_function** (likely an address in the .text segment).

Exercise 2 - Anatomy of Memory- Code Is Data

```c
typedef void (*func_t)();
void secret_function() { printf("Win!\n"); }

void trigger() {
    // 1. Allocate space for a function pointer on the HEAP
    func_t *heap_hook = malloc(sizeof(func_t));

    // 2. Store the address of the code into that heap memory
    *heap_hook = secret_function;

    // <-- SNAPSHOT TAKEN HERE
    free(heap_hook);
}
int main() {
    trigger();
    return 0;   // Line 16
}
```

Q2 The ABI/Stack Frame: When `main` calls `trigger()`, the system will push a value onto the stack (RISC-V/x86-64. What specifically is that value?

Q1: When trigger is called, the CPU pushes the Return Address (Instruction Pointer) onto the stack.

This value is the memory address of the instruction immediately following the call (Line 16).

This implies that a **Code Pointer** (the Return Address) sits on the stack, right near the local variables of trigger.

This is the fundamental mechanism exploits like Buffer Overflows use to hijack control flow that we will see later.

A look at the objdump (RISC-V)

```
0000000000000724 <secret_function>:
 724:   1141                    addi    sp,sp,-16
 726:   e406                    sd  ra,8(sp)
    ... # Call printf("Win!\n");
 73e:   0141                    addi    sp,sp,16
 740:   8082                    ret


0000000000000776 <main>:
 776:   ...

 77e:   fc5ff0ef                jal 742 <trigger>
 782:   4781                    li  a5,0  # Write the 0 in a register for return
 784:   853e                    mv  a0,a5
 ...
 78c:   8082                    ret
```

riscv64-linux-gnu-gcc test.c -o test.O # I get an object file
riscv64-linux-gnu-objdump -D test.O > dump.S # I get the readable “objdump” showing me the assembly with all the addresses

The Step-by-Step Breakdown:

1. The Call from main (Setting the Stage)
Location: 0x77e inside main.
Instruction: jal 742 <trigger>
What happens: The CPU jumps to 0x742. It sets the **Return Address Register (ra)** to the *next* instruction: **0x782**

2. The trigger Prologue (Saving the Return Address)
Location: 0x742 - 0x744 inside trigger.
Instruction: addi sp,sp,-32 followed by **sd ra,24(sp)**.
The ABI Moment: The function claims 32 bytes of stack. Immediately, it takes the value 0x782 (currently in register ra) and saves it to memory at address sp + 24.

> *Why?* Because trigger calls malloc later. The ra register will be overwritten by that call, so the original return address (0x782) must be preserved in RAM (on the stack).

3. The Heap Allocation (malloc)
Location: 0x74c - 0x750.
Action: malloc is called. It returns a memory address (let's pretend it returns 0xHEAP_100) in register a0.
Storage: sd a5,-24(s0) (at 0x752) saves this heap pointer into the stack frame local variables.

Note: s0 is the frame pointer. -24(s0) translates to sp + 8. So, your local variable heap_hook lives at sp + 8.

4. The "Double Indirection" (Very tricky to understand, all that just to produce a constant)
Location: 0x75a - 0x762.
Calculation: auipc and addi work together to calculate the address of secret_function.

Result: 0x724 (The start of secret_function).

The Write: sd a4,0(a5)

This writes the value 0x724 (Text Segment Address) **into** the memory at 0xHEAP_100 (Heap Segment).

Complete ASM snippet:

```
0000000000000742 <trigger>:
 742:   1101                addi    sp,sp,-32
 744:   ec06                sd  ra,24(sp)
 746:   e822                sd  s0,16(sp)
 748:   1000                addi    s0,sp,32
 74a:   4521                li  a0,8
 74c:   ec5ff0ef          jal 610 <malloc@plt>
 750:   87aa                mv  a5,a0
 752:   fef43423          sd  a5,-24(s0) # ffffffffffffcfe8 <__global_pointer$+0xfffffffffffa7e8>
 756:   fe843783          ld  a5,-24(s0)
 75a:   00000717           auipc   a4,0x0
 75e:   fca70713          addi    a4,a4,-54 # 724 <secret_function>
 762:   e398                sd  a4,0(a5)
 764:   fe843503          ld  a0,-24(s0)
 768:   ec9ff0ef          jal 630 <free@plt>
 76c:   0001                nop
 76e:   60e2                ld  ra,24(sp)
 770:   6442                ld  s0,16(sp)
 772:   6105                addi    sp,sp,32
 774:   8082                ret


0000000000000776 <main>:
 776:   1141                addi    sp,sp,-16
 778:   e406                sd  ra,8(sp)
```

```
77a:   e022              sd  s0,0(sp)
77c:   0800              addi    s0,sp,16
77e:   fc5ff0ef          jal 742 <trigger>
782:   4781              li  a5,0
784:   853e              mv  a0,a5
786:   60a2              ld  ra,8(sp)
788:   6402              ld  s0,0(sp)
78a:   0141              addi    sp,sp,16
78c:   8082              ret
```

A look at the objdump (RISC-V)

```
0000000000000776 <main>:
 776:   ...

 77e:   fc5ff0ef                jal 742 <trigger>
 782:   4781                    li  a5,0  # Write the 0 in a register for return
 784:   853e                    mv  a0,a5
...
 78c:   8082                    ret


0000000000000742 <trigger>:
 742:   1101                     addi    sp,sp,-32  # Reserve 32 bytes for the stack
 744:   ec06                     sd  ra,24(sp) #Save the returned address on the stack
 ...
 74a:   4521                     li  a0,8
 74c:   ec5ff0ef                 jal 610 <malloc@plt> # We call malloc for 8 bytes
 ... # We put the number 724 <secret_function> inside register a4
 762:   e398                     sd  a4,0(a0) # We store address 724 where malloc said.
 ...
 768:   ec9ff0ef                 jal 630 <free@plt> # Calling free
 ...
 76e:   60e2                     ld  ra,24(sp) # Restore the return address from the stac
 770:   6442                     ld  s0,16(sp)
 772:   6105                     addi    sp,sp,32 # Release the stack
 774:   8082                     ret
```

riscv64-linux-gnu-gcc test.c -o test.O # I get an object file
riscv64-linux-gnu-objdump -D test.O > dump.S # I get the readable “objdump” showing me the assembly with all the addresses

The Step-by-Step Breakdown:

1. The Call from main (Setting the Stage)
Location: 0x77e inside main.
Instruction: jal 742 <trigger>
What happens: The CPU jumps to 0x742. It sets the **Return Address Register (ra)** to the *next* instruction: **0x782**

2. The trigger Prologue (Saving the Return Address)
Location: 0x742 - 0x744 inside trigger.
Instruction: addi sp,sp,-32 followed by **sd ra,24(sp)**.
The ABI Moment: The function claims 32 bytes of stack. Immediately, it takes the value 0x782 (currently in register ra) and saves it to memory at address sp + 24.

> *Why?* Because trigger calls malloc later. The ra register will be overwritten by that call, so the original return address (0x782) must be preserved in RAM (on the stack).

3. The Heap Allocation (malloc)
Location: 0x74c - 0x750.
Action: malloc is called. It returns a memory address (let's pretend it returns 0xHEAP_100) in register a0.
Storage: sd a5,-24(s0) (at 0x752) saves this heap pointer into the stack frame local variables.

> *Note:* s0 is the frame pointer. -24(s0) translates to sp + 8. So, your local variable heap_hook lives at sp + 8.

4. The "Double Indirection" (Very tricky to understand, all that just to produce a constant)
Location: 0x75a - 0x762.
Calculation: auipc and addi work together to calculate the address of secret_function.

> Result: 0x724 (The start of secret_function).

The Write: sd a4,0(a5)

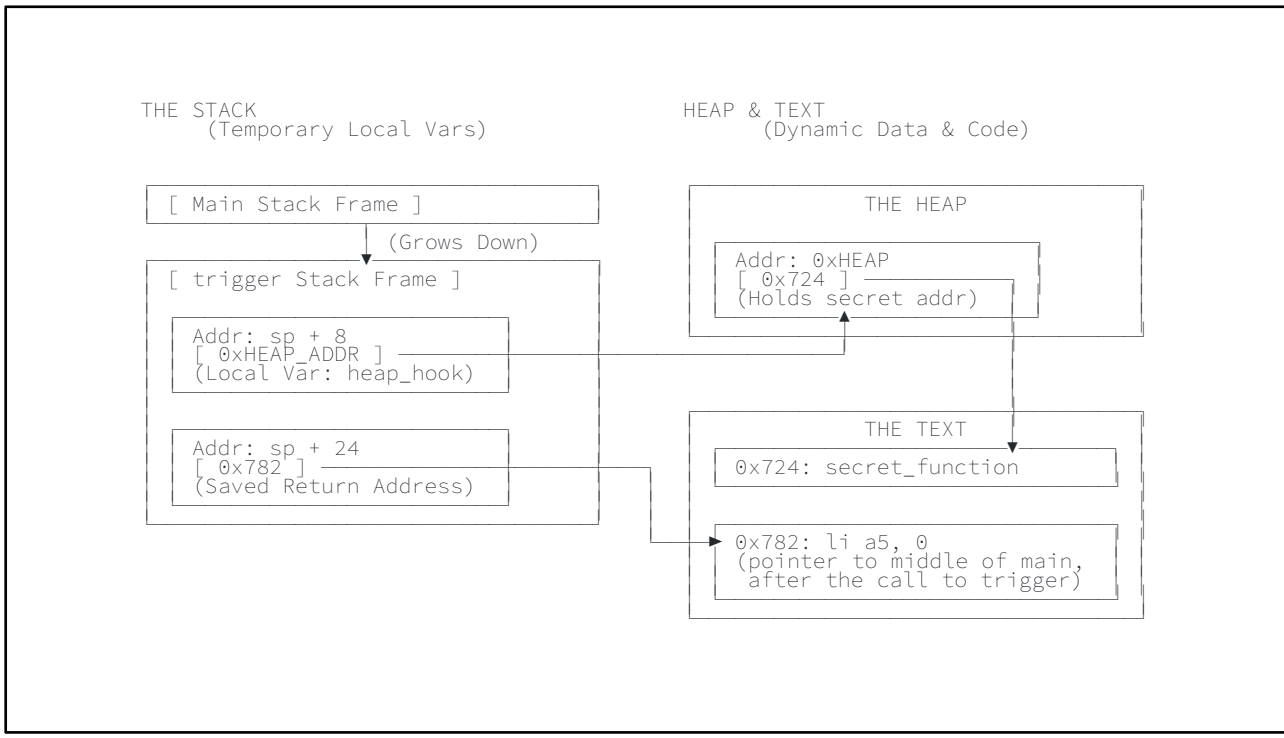
> This writes the value 0x724 (Text Segment Address) **into** the memory at 0xHEAP_100 (Heap Segment).

Complete ASM snippet:

```
0000000000000742 <trigger>:
 742:   1101                addi    sp,sp,-32
 744:   ec06                sd  ra,24(sp)
 746:   e822                sd  s0,16(sp)
 748:   1000                addi    s0,sp,32
 74a:   4521                li  a0,8
 74c:   ec5ff0ef          jal 610 <malloc@plt>
 750:   87aa                mv  a5,a0
 752:   fef43423          sd  a5,-24(s0) # ffffffffffffcfe8 <__global_pointer$+0xfffffffffffa7e8>
 756:   fe843783          ld  a5,-24(s0)
 75a:   00000717           auipc   a4,0x0
 75e:   fca70713          addi    a4,a4,-54 # 724 <secret_function>
 762:   e398                sd  a4,0(a5)
 764:   fe843503          ld  a0,-24(s0)
 768:   ec9ff0ef          jal 630 <free@plt>
 76c:   0001                nop
 76e:   60e2                ld  ra,24(sp)
 770:   6442                ld  s0,16(sp)
 772:   6105                addi    sp,sp,32
 774:   8082                ret


0000000000000776 <main>:
 776:   1141                addi    sp,sp,-16
 778:   e406                sd  ra,8(sp)
```

```
77a:   e022              sd  s0,0(sp)
77c:   0800              addi    s0,sp,16
77e:   fc5ff0ef          jal 742 <trigger>
782:   4781              li  a5,0
784:   853e              mv  a0,a5
786:   60a2              ld  ra,8(sp)
788:   6402              ld  s0,0(sp)
78a:   0141              addi    sp,sp,16
78c:   8082              ret
```


THE STACK
(Temporary Local Vars)
HEAP & TEXT
(Dynamic Data & Code)
[Main Stack Frame]
(Grows Down)
[trigger Stack Frame]
Addr: sp + 8
[0xHEAP_ADDR]
(Local Var: heap_hook)
Addr: sp + 24
[0x782]
(Saved Return Address)
THE HEAP
Addr: 0xHEAP
[0x724]
(Holds secret addr)
THE TEXT
0x724: secret_function
0x782: li a5, 0
(pointer to middle of main,
after the call to trigger)

Computer Security (COM-301)

Software security

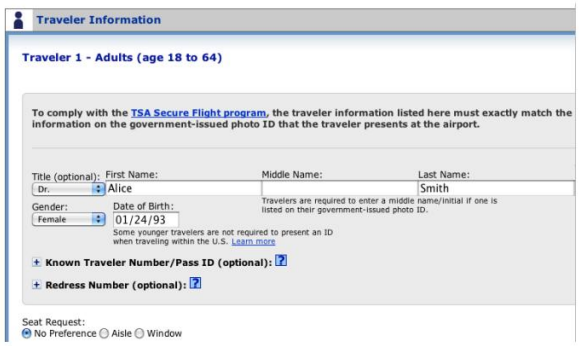
Memory safety

Slides by Carmela Troncoso
Some slides/ideas adapted from: Tuomas Aura, Yoshi Kohno, Trent Jaeger

Why all the fuzz with overflows…

(This is an example for the purpose of illustrating the damage that an overflow – i.e., writing on a variable past its allocated length – can produce. Any resemblance with reality is pure coincidence)

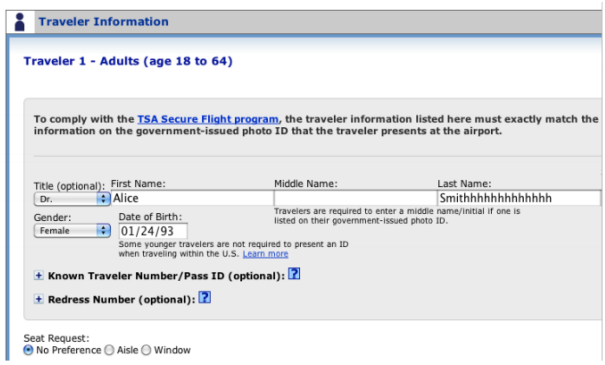
Imagine a simple check-in form in which users of an airline input their data, and these data are shown to the desk staff when they are receiving their boarding passes.

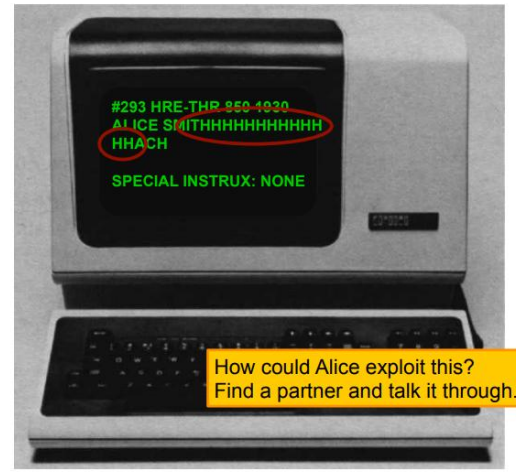


This form takes the information shown in the figure on the top left: the passenger First, Middle and Last name, and the passenger's Gender and Date of Birth.

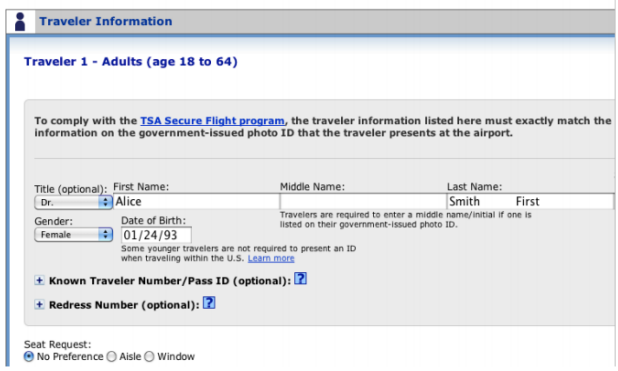
None of this information is sanitized (in particular, checked for length) when stored in the server

At the airport, this information is shown to the desk staff with the following format (see bottom right figure):
Line 1: some internal information about the flight number
Line 2: name of the passenger (extracted from the information the passenger provides in the form)
Line 3: ticket type (first class, business, coach, etc)
Line 4: [blank]
Line 5: Special requisites for this passenger formatted as the string "Special Instrux:" followed by the requisites – None if the passenger does not have any special treatment.



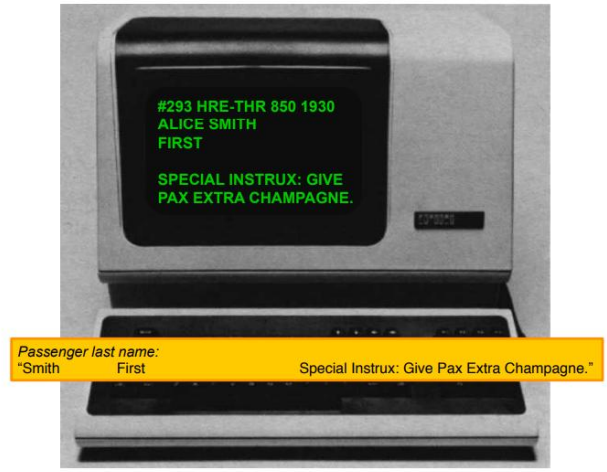


The old screen at the desk allows 21 characters per line. After that, it starts overwriting the next line, as shown in the bottom right.



If Alice knows that there is no length check, and also the configuration of the screen, instead of writing random characters she can overwrite the second line with something clever.

In particular if after her Last Name she writes 10 spaces, the next information will be written in the next line (line 3, where the ticket type appears)



In fact given this knowledge Alice can go further and overwrite also the special instruction by introducing the adequate number of spaces after the ticket type.

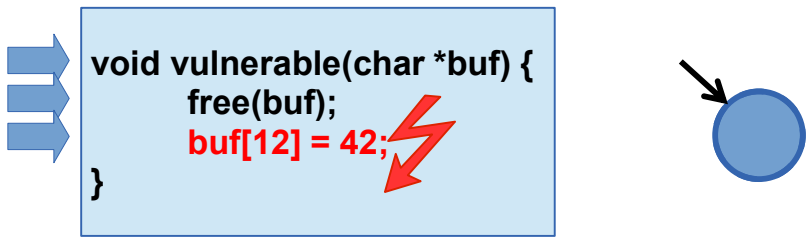
Memory corruption

Unintended modification of memory location due to missing / faulty safety check

```
void vulnerable(int user1, int *array) {
        array[user1] = 42;
}
```

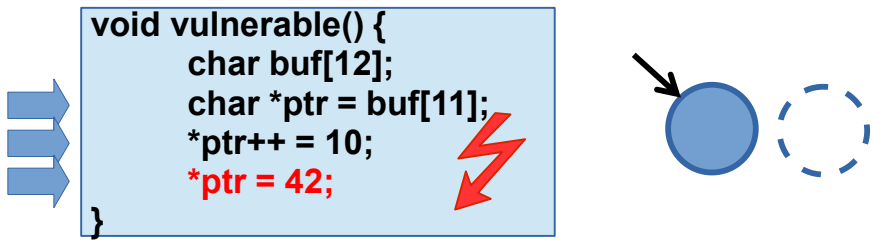
Memory corruption happens when a region of the memory that *is not allocated to a program* is modified by this program. The C language does not check for this situation, so it can happen when the programmer misses a check, or does not check for all possible cases.

Memory safety: temporal error



Temporal safety when accessing an object means that the pointed-to object is the same as when the pointer was created. When an object is freed (e.g., by calling free for heap objects or by returning from a function for stack objects), the underlying memory is no longer associated to the object and the pointer is no longer valid.
Accessing the region of memory pointed by such an invalid pointer results in a ***temporal memory safety*** error and undefined behavior.

Memory safety: spatial error



Spatial memory safety is a property that ensures that all memory accesses in a program are within the bounds of their pointers valid objects. A pointer references a specific address in an application's address space. Memory objects are allocated explicitly by calling into the memory allocator (e.g., through malloc) or implicitly by calling a function for local variables. An object's bounds are defined when the object is allocated and a pointer to the object is returned.

Accessing memory using a pointer that points outside of the associated object results in a ***spatial memory safety error*** and undefined behavior.

In many cases a spatial memory safety error can result on a **segmentation fault** that causes the program to stop (see more about what is a segmentation fault and what causes it here: https://stackoverflow.com/questions/2346806/what-is-a-segmentation-fault)

Memory safety: spatial error - variation

Variable that stores whether the user is authenticated to call a function that reads secrets

```
void vulnerable()
{
int authenticated = 0;
char buf[80];

gets(buf);
...
}
```

How can you exploit this?

If we give more than 80 characters from stdin, it will overwrite `authenticated`! *(both are in the stack)*

If the value is !=0 the user will be authenticated!

`Gets(buf)`: reads a line from stdin and stores it into the string pointed to by `buf`

Here is an example of a problem that can happen when the boundaries of the allocated memory are not checked.
The function `gets` just reads anything that the user inputs. However, it does not check the boundary of the memory reserved to `buf`.

If the value input by the user is too long, it may overwrite `authenticated` (which is stored), causing problems later when the program checks the `authenticated` value as if it has been modified and is different from zero the user will be considered as authenticated

[Note: This may **not** work on your computer as is, as it depends on the protections your OS has implemented, and on the concrete architecture that determines the order in which variables are stored on a function's stack frame.
If you are interested in more Stack manipulations, you can learn about it in COM-402 at the masters]

Exercise: Danger of null-terminated strings

```c
int main(int argc, char** argv) {
    char buffer[10];
    char secretData[60];
    if (argc < 2) { exit(1); }

    strcpy(secretData, "donkeysAreTheCoolestAnimal");
    strncpy(buffer, argv[1], 10);
    printf(buffer);
    return 0;
}
```

Question 1: What does `./myProgram` do? What does `./myProgram Hello` do?

Question 2: Can I craft a clever argument to call my program, that will make it print the secretData?

Question 1
What does ./myProgram do?
It exits immediately.
Why? When you run a program without arguments, argc (argument count) is 1 (just the program name). The line if (argc < 2) evaluates to true, so the program executes exit(1) and terminates before doing anything else.

What does ./myProgram Hello do?
It prints "Hello".
Why?

argc is 2, so it passes the check.
The string "Hello" is 5 characters long.
strncpy(buffer, argv[1], 10) copies the 5 characters of "Hello" into the buffer.
Because the string is shorter than the limit (10), strncpy fills the remaining 5 bytes of the buffer with **null terminators (\0)**.
printf(buffer) reads "Hello", hits the first null terminator, and stops printing.

Question 2:
Yes. (Assuming the compiler places secretData directly after buffer in memory).
You need to pass a string that is exactly 10 characters long (or longer). Example:
./myProgram 0123456789

This exploits the specific behavior of strncpy when the source is equal to or larger than the size limit.
strncpy(buffer, input, 10) copies exactly 10 bytes. Because the input filled the count completely, strncpy does NOT write a null terminator (\0) at the end.

The Memory State:

- buffer is now completely full of characters: ['0', '1', ... '9'].
- The very next byte in memory is the start of secretData (assuming standard stack layout for this exercise).
- The "wall" (the null terminator) that usually separates buffer from secretData is gone.

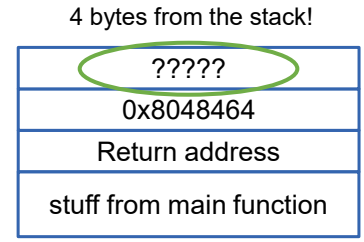
When printf(buffer) runs, it starts printing the numbers. It reaches the end of buffer but finds no \0 to tell it to stop. It blindly continues reading into the next memory address, which holds secretData, and prints "donkeysAreCoolestAnimal" until it finally hits the natural null terminator at the end of the secret string.

Uncontrolled Format String *(CWE-134)*

What would this print if `argv[1] = "You scored %d\n"`?

```
#include<stdio.h>
int main(int argc, char** argv) {
char buffer[100];

strncpy(buffer, argv[1], 100);
printf(buffer);
return 0;
}
```



And if it was printf("`You scored %d %d %d %d`")?
And if it was printf("`You scored %s`")?

Format string **can read** beyond the parameters
e.g, if input = '%4$p" → Read from 4th parameter (even if it does not exist)

Format string **can write** to memory
e.g, if input = '%6$n" → Write to the address pointed to by 6th parameter

http://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html
https://owasp.org/www-community/attacks/Format_string_attack

In the code in the example, we are getting a string to print as an argument. The string can be anything. If in the string given as input there is a parameter, then it will be interpreted that the value of this parameter is in the next bytes of the stack (the number of bytes will be given by the specifier used in the string.

Note that there could be another thing we could do: pass call char .

- %d, which prints and int, will print 4 positions from the stack
- %s, which prints a string, will print until it finds '\0' the character that indicates end of string.

Note that particular format specifiers allow to read and write from positions beyond the next in the stack

```
#include<stdio.h>
int main(int argc, char** argv) {
char buffer[100];
strncpy(buffer, argv[1], 100);
printf("%s", buffer);
return 0;
}
```

SOLVING THE PROBLEM

The programmer should decide the format of the string. That ensures that no extra argument, read or write, can be used.

27

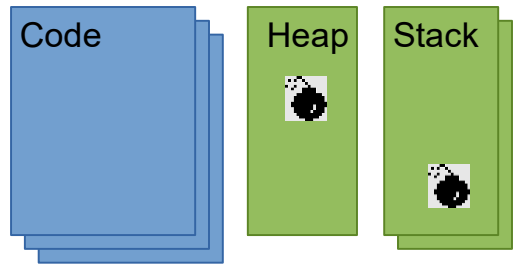
http://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html
https://owasp.org/www-community/attacks/Format_string_attack

In order to avoid this problem, it is very important that the programmer defines the parameters, and does not let the user input them.

Attack scenario: code injection

Force memory corruption to set up attack

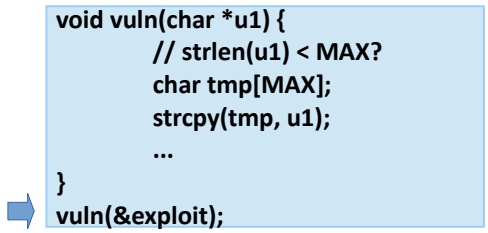
Redirect control-flow to injected code



The goal of a code injection attack goal is to execute code (e.g. access a file) into a running process or modifying the program flow to execute unexpected commands. The means in injecting new code.

Control flow attacks most common on current systems. In these attacks the adversary uses memory corruption to modify a code pointer and prepare data to be processed by system functions.

Code injection attack

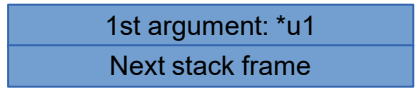


When a function is called, the program prepares the stack.

It reserves a new stack frame where the data of the function will be store

Code injection attack

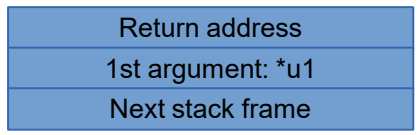
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```



First, the argument of the function is pushed to the stack

Code injection attack

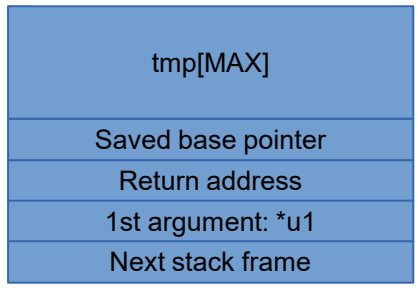
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```



Second, the return address for the program (where to go after the current function is finished) is pushed to the stack.

Code injection attack

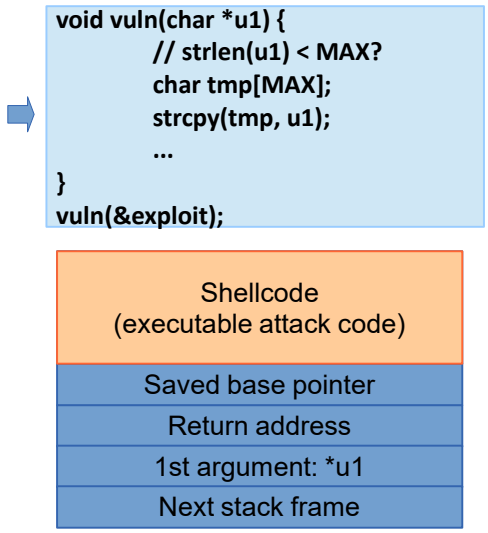
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```



Before starting the function, we also reserve space for the local variables in the stack, in this case `MAX` bytes for the variable `tmp`.

There is also saved space for the so-called Base pointer (4 bytes in 32-bit operative systems / 8 bytes in 64-bit operative systems), which is irrelevant for this lecture.

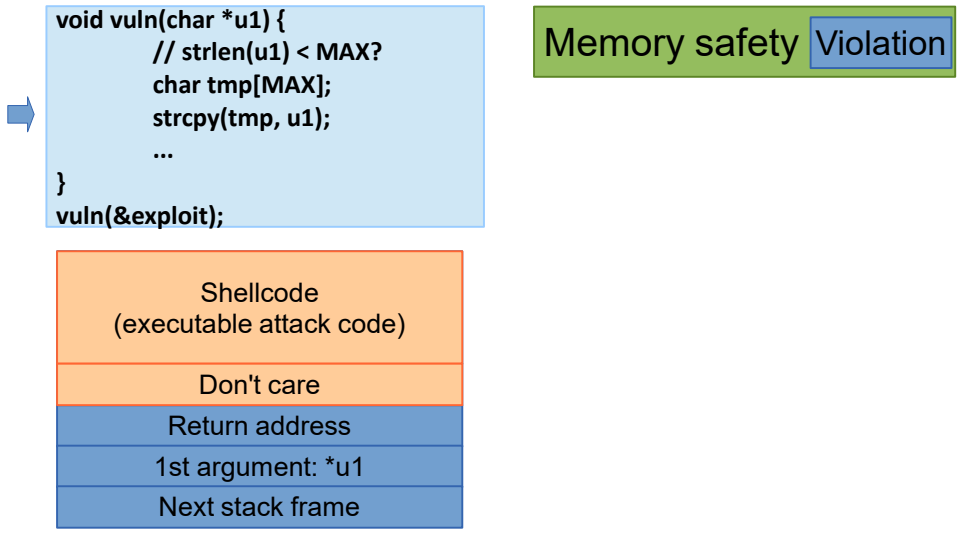
Code injection attack



When executing `strcpy`, the program will start copying the content of `u1` into `tmp`.

Let us consider that the content of u1 is some executable code that implements an attack.

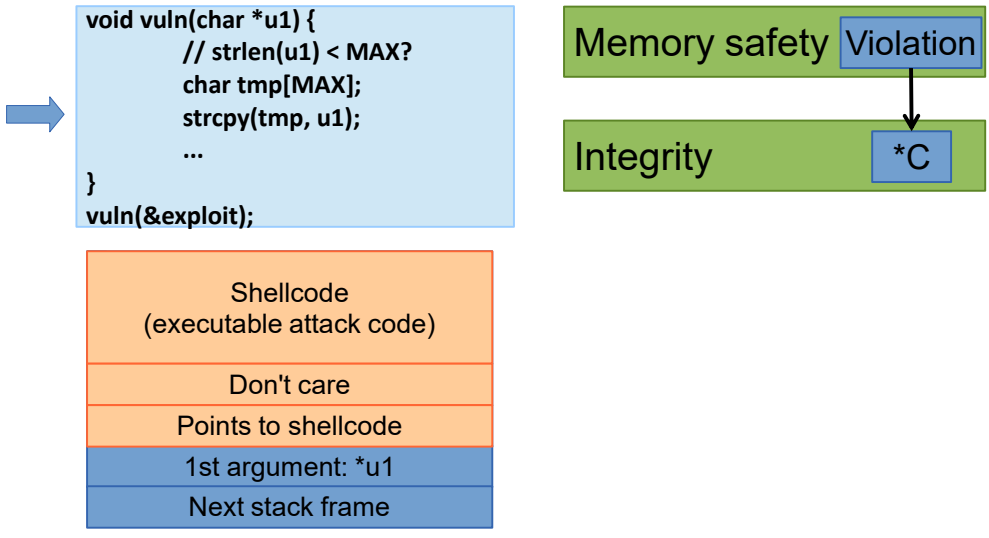
Code injection attack



As there is no control on the size of u1, if this variable is longer than MAX bytes, it will overwrite the next value in the stack, the base pointer. (We do not care what value is written there as it will not be used)

At this point, there is a pointer that may point to memory that is no allocated for the program variable: there is a **memory safety violation**.

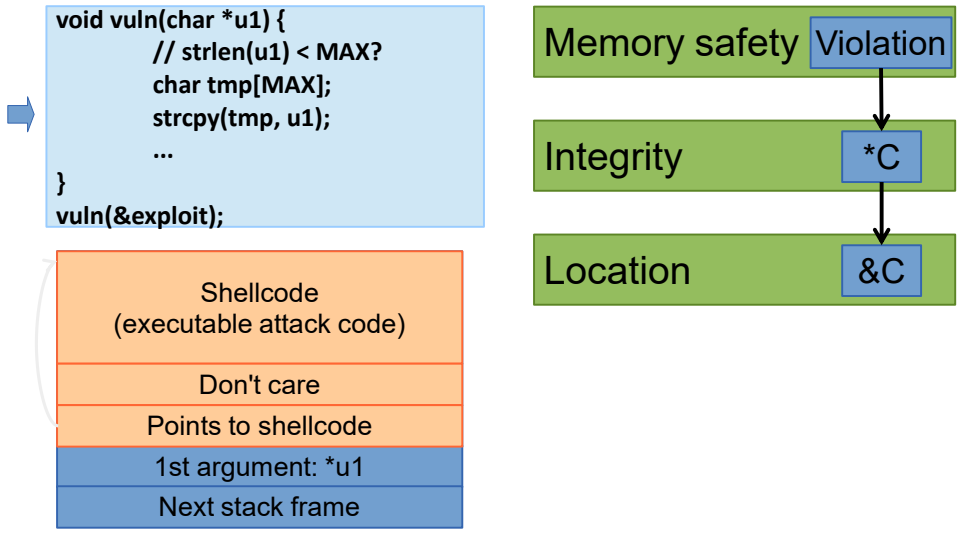
Code injection attack



If the content of u1 is even longer, the program continues writing and will overwrite the return address.

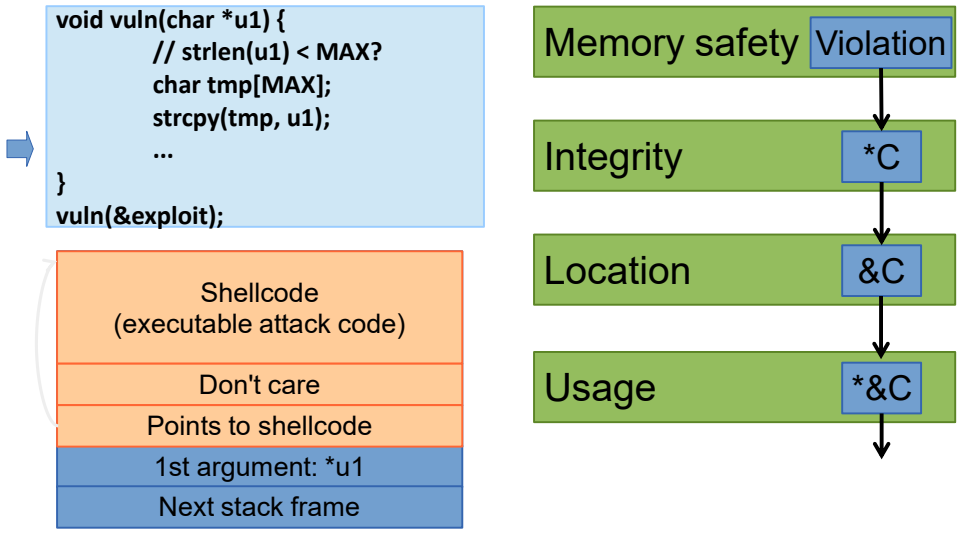
At this point the program has violated the **integrity** of the return address pointer.

Code injection attack



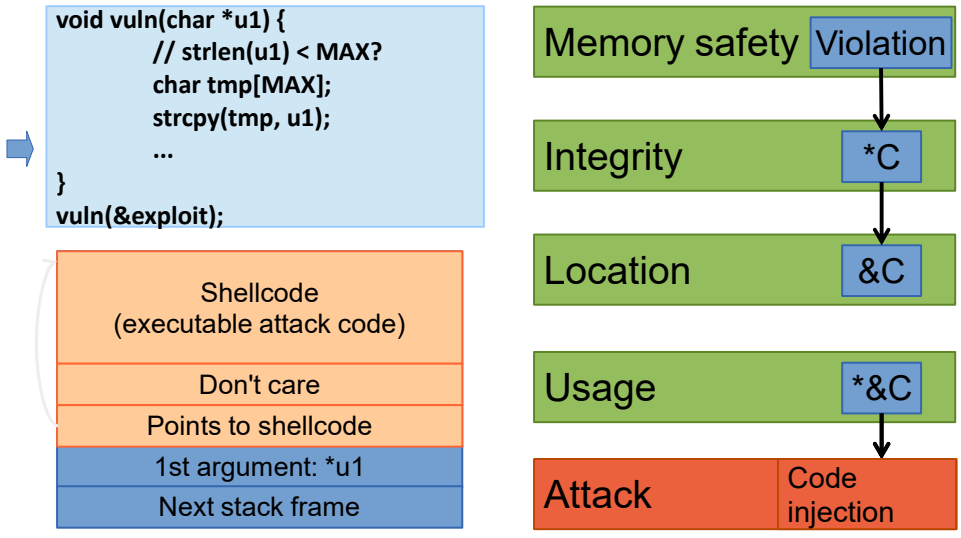
The return address is overwritten with a new address: the address where the executable attack code start (we change the **location** where the execution will go next)

Code injection attack



When the function ends, the program will **use** the corrupted return address to continue the program, i.e., the attack code.

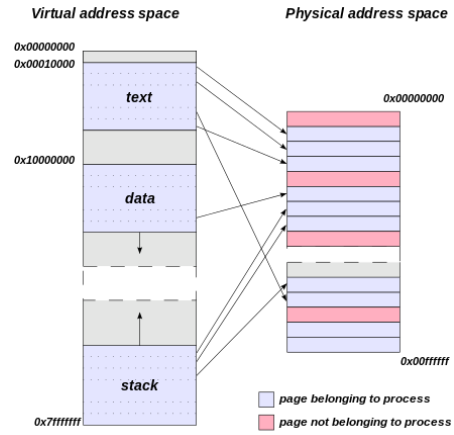
Code injection attack



At this point the adversary has succeeded in their attack: they can execute arbitrary code!

Data Execution Prevention

- Enforces code integrity on page granularity
  - Execute code if eXecutable bit set
- W^X ensures write access or executable
  - Mitigates against code corruption attacks
  - Low overhead, hardware enforced, widely deployed
- Weaknesses and limitations
  - No-self modifying code supported

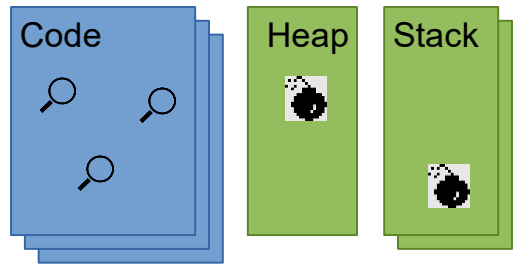


A defense against code injection is **Data Execution Prevention (DEP)**. This is a countermeasure enforced at the hardware level. It protects the memory at a page granularity. Every page on a program is assigned as writable **OR** executable. Thus, the stack, where the adversary can write, can never be executed.

A limitation of this countermeasure is that it prevents self-modifying code. This prevents many functionalities in applications offered as a service, where the user executes code supplied by the server on their machine (e.g., Javascript being executed on the browser).

Attack scenario: code reuse

- Find addresses of gadgets
- Force memory corruption to set up attack
- Redirect control-flow to gadget chain



In a code injection attack, the adversary first writes code, and then gets the OS to execute this code.
If DEP is in place, however, executing writable memory becomes impossible.
Thus, this attack cannot be deployed.

To circumvent this protection, instead of executing injected code, the adversary can find pieces of code already that already exist in memory (and therefore are executable) and redirect the program flow to those pieces.
These pieces are typically known as *gadgets*.

Control-flow hijack attack

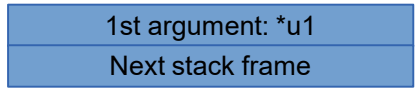
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        …
}
vuln(&exploit);
```

Next stack frame

The attack starts as a code injection attack, when the OS prepares the stack for the function call.

Control-flow hijack attack

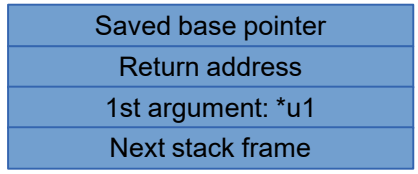
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```



The OS reserves space for the function argument

Control-flow hijack attack

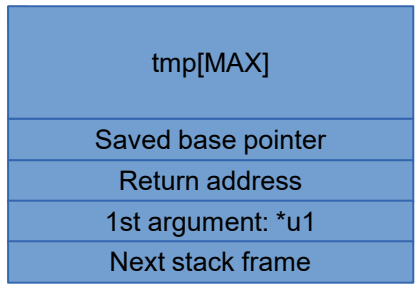
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```



The OS then reserves space for the return address and the base pointer

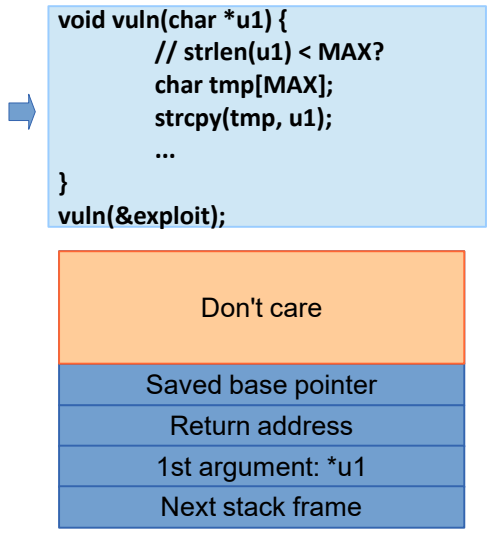
Control-flow hijack attack

```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```



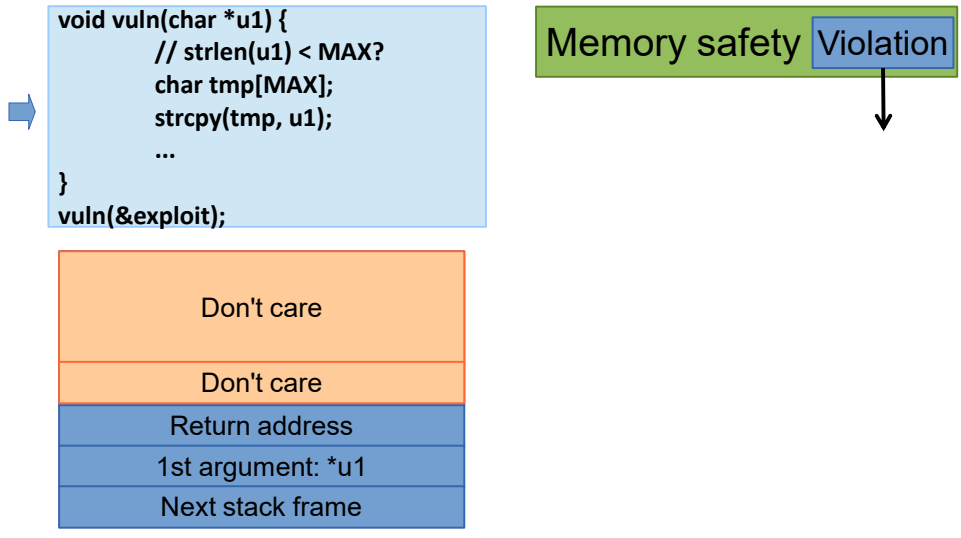
And finally space for the variable tmp inside of the function

Control-flow hijack attack



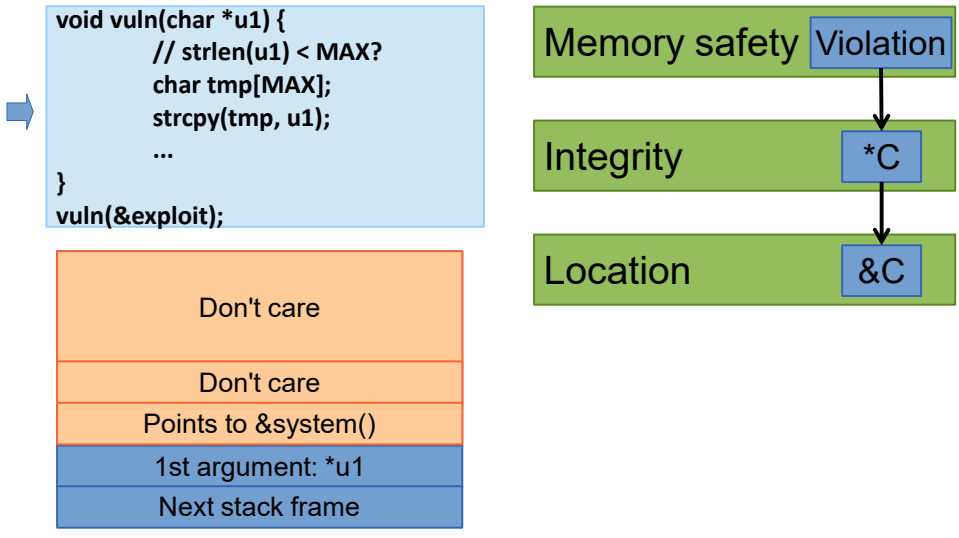
The adversary exploits the same lack of check as in the code injection attack to write beyond the boundaries of `tmp`.
However, as it is not possible to execute the code in the stack, the adversary now does not care about what is written in the first `MAX` bytes: this code will not be executed.

Control-flow hijack attack



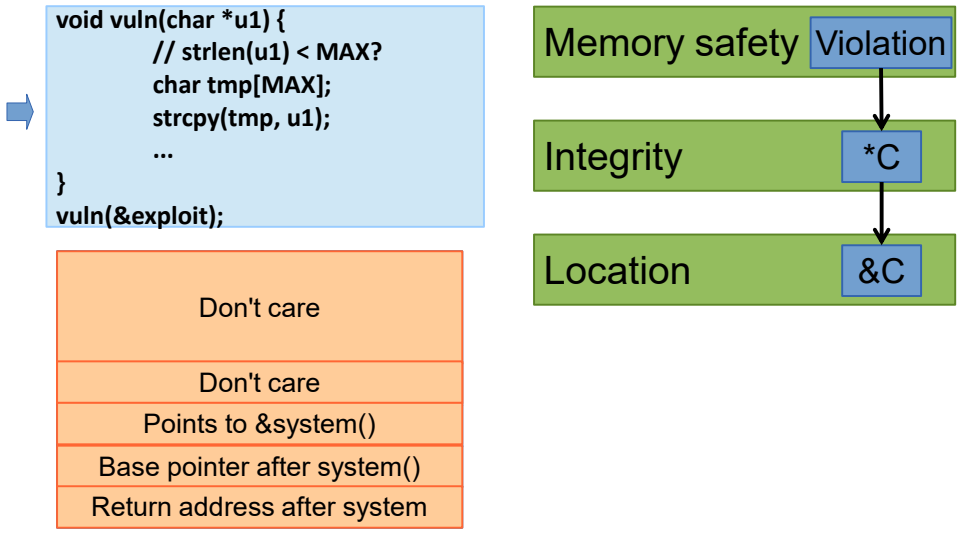
As before, a **memory safety** violation happens as soon as the adversary overwrites pointers that they are not allowed to write on.

Control-flow hijack attack



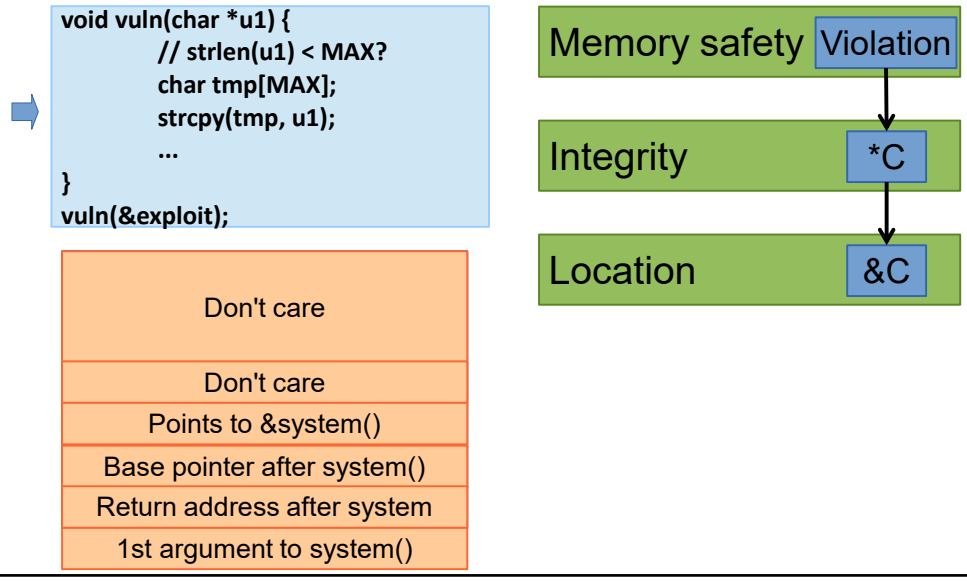
But now, instead of point to the start address of `tmp`, as it would happen in code injection, the adversary modifies the address pointed by the return pointer to be the location of an executable function somewhere in the memory, e.g., the `system`() function

Control-flow hijack attack



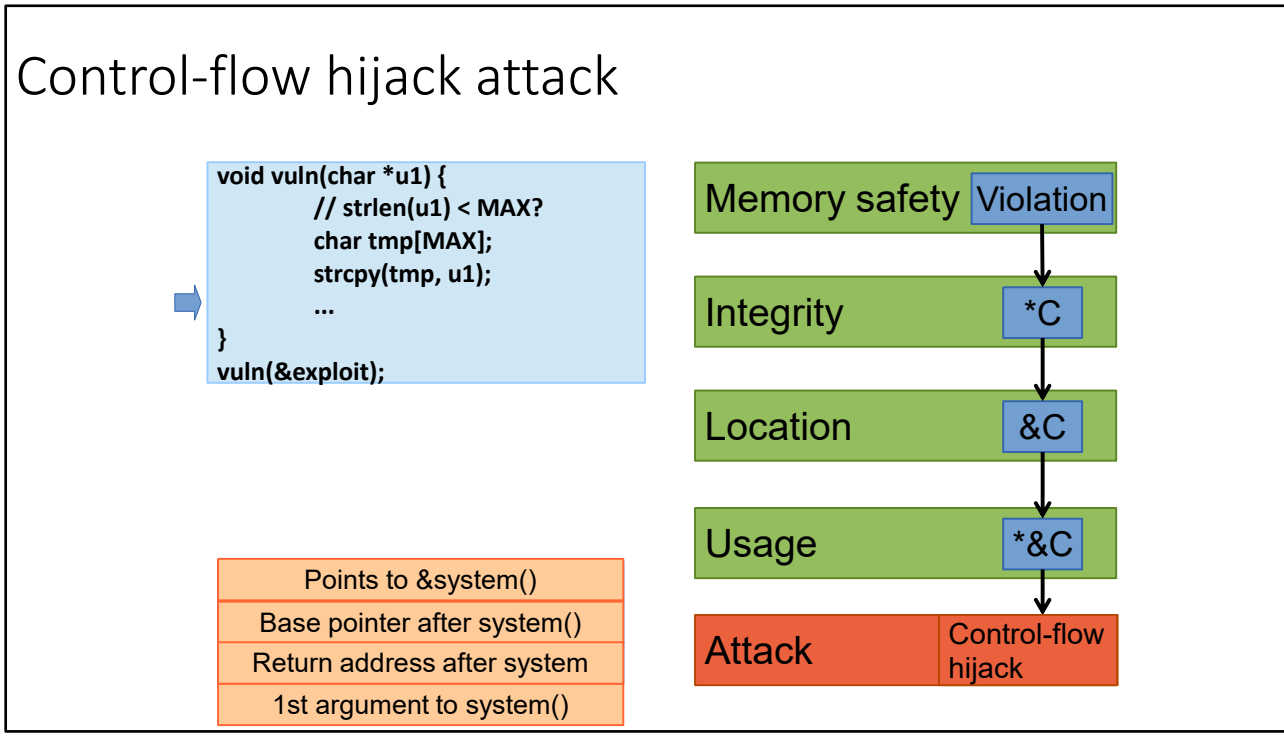
As the adversary prepares to return to the new location (system()), they also need to prepares the stack to be in the state that `system`() is expecting: they need to add the base pointer, the return address after system() is called…

Control-flow hijack attack



… and then the argument that system() will receive, i.e., the command that will be executed

Control-flow hijack attack



When the function `vuln`() ends, the program will continue its flow to `system`().
At this point in time, the adversary has hijacked the flow of the program to redirect to where they want.

Typically, the adversary will try to use several gadgets in a row by exploiting bugs in different functions in order to be able to execute arbitrary chains of instructions.

First Defense: Address Space Layout Randomization

- **Goal**: prevent the attack from reaching a target address
- Randomizes locations of code and data regions
  - Probabilistic defense
  - Depends on loader and OS
- Weaknesses and limitations
  - Undefined behavior: prone to information leaks
  - Some regions remain static (on x86)
  - Performance impact? (Small on modern machines, bigger on older machines)

Hijack attacks are enables by the fact that the adversary knows where system functions reside in memory.

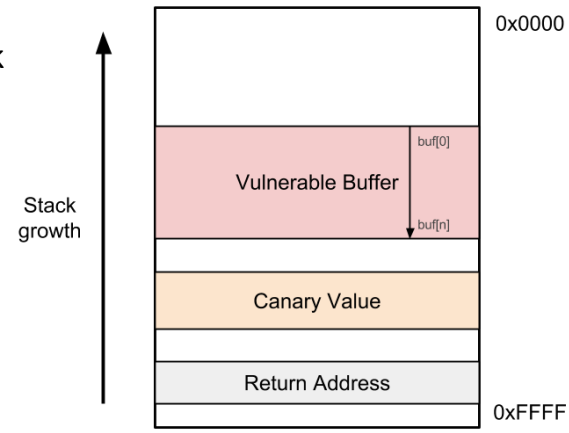
A defense to avoid these attacks is to *randomize* the memory layout so that the adversary *cannot* know a priori where to redirect the function thus reducing the likelihood of the attack. This randomization depends on the capabilities of the operative system and the loader that maps OS functions in memory. As such, the defense is probabilistic. The adversary does not know where functions reside, but can guess (with lower or higher probability depending on the randomization implemented).

This defense has the following problems:
- The adversary can still redirect the program. This does not guarantee success of the attack, but also does not guarantee that nothing bad will happen. The adversary may end up triggering other unintended functionality and reading from memory.
- In runtime the OS needs to “undo” the randomization to execute the program. This can slow down execution.
- Not all regions can be randomized. Sregions are always the same and ASLR do not defend them.

Reminescent of “Recording Compromise” –
Record Attempts of Attacks!

Stack canaries

- Protect return instruction pointer on stack
  - Compiler modifies stack layout
  - Probabilistic protection
- Weaknesses and limitations
  - Prone to information leaks
  - No protection against targeted writes / reads



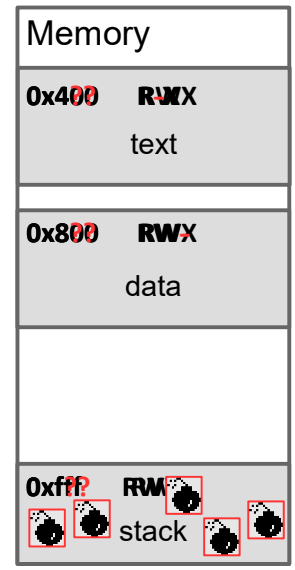
Stack canaries, like coal mine canaries, are a means to detect if something fishy is going on. Stack canaries are a value between the part of the stack writable by the program and the return address. The idea is that, if the canary value has changed (the canary 'has died') then it is not safe to use the return address, as it could have been compromised.

This mitigation is also probabilistic, in the sense that the adversary may be able to predict the canary and overwrite it.

Also, the fact that the adversary cannot write on the stack beyond the canary, does not mean she cannot read (e.g., exploiting an uncontrolled format string vulnerability). As such, it cannot prevent information leaks. Also, canaries do not protect against vulnerability exploits that can target a particular address.

Status of deployed defenses

- Data Execution Prevention (DEP)
- Address Space Layout Randomization (ASLR)
- Stack canaries
- Safe exception handlers
  - Pre-defined set of handler addresses



The currently deployed defenses work as follows. (The three first are explained in the previous slides.)

- DEP: protects the memory making sure that the text part of the memory (i.e., the original program) is executable but cannot be overwritten; and that the data can be written but not executed.

- ASLR acts across the memory, effectively scrambling addresses (i.e., the two last bytes of the address are unknown to the adversary)

- Stack canaries are inserted in the stack, helping to detect overflow attacks.

- Windows also uses safe exception handlers, which aim at keeping the system safe even after errors. This countermeasure makes sure that, after an error there is no undefined behavior, but the system only can execute a pre-defined set of error handling functions.

Software testing

Testing is the process of executing a program to find errors

Error: deviation between observed behavior and specified behavior (a violation of the underlying specification)

- Functional requirements
- Operational requirements
- Security requirements?

Software testing executes code under different circumstances with the goal of finding configurations that raise an *error*. An error is a deviation between how we expect the program would function and what actually happens. This can be:
- an error regarding functionality: the program does not provide the expected result
- and error regarding operation: the program crashes, is too slow (even never terminating)

But what about security? Testing for security is hard. We cannot ensure that we have found *all* bugs that matter (the adversary will do things that have not or cannot be tested). Therefore, we cannot prove the absence of security-critical bugs using testing. Still, finding as many bugs as possible helps increasing the safety of software.

Security testing

> "Testing can only show the presence of bugs, never their absence."
> (Edsger W. Dijkstra)

Complete testing of all
- Control-flows: test all path through the program
- Data-flow: test all values used at each location

Practical testing is limited by state explosion

Ideally, we would like to test all possible
Control-flows: all possible paths through the program, i.e., all possible outcomes of branches in a program (if-else clauses, for clauses, while clauses, etc).
Data-flows: all possible values for the variables / locations that are used by the program.

Of course, testing *all* possible paths and data values is impossible in general.

Control-Flow vs. Data-Flow

```
void program() {
        int a = read();
        int x[100] = read();

        if (a >=0 && a <= 100) {
                x[a] = 42;
        }
        ...
}
```

59

The difference between **control-flow** and **data-flow".**
Consider this example program.

The values a=12 and a=101 cover all flows:
When a= 12, (a>=0 && a<=100) is True, and the instruction within the if (x[a]=42) is executed.
When a=101, (a>=0 && a<=100) is False and the instruction within the if is not executed.

However, even all statements are executed, and both flows are explored, not all data-flows are considered, i.e., we did not consider all possible values of variables in each instruction.

This may be very relevant. For instance, for this program, the data-flow a=100 would raise a bug. In this case (a>=0 && a<=100) would be True, but x[100] is not reserved for the program (x has 100 positions starting in position 0). Thus, when arriving to the instruction x[a] = 42, the program would crash trying to access x[100]

How to test security properties

Manual Testing: testing is designed by a human
- Heuristic test cases

Code Reviews

Automated testing: testing is decided algorithmically
- Algorithms designed to run the program and find bugs
- Algorithms enhanced by means to enforce properties

There are two ways of testing for security properties:

Manual testing in which the tests to be carried out are defined by a human, trying to identify corner cases that may appear in reality. Whether these corner cases could trigger a bug is typically done via *code reviews*, in which humans read each others' code to search for programming errors; or by implementing *test cases* so that the checks can be performed in runtime.

Automated testing in which the tests are not defined directly by a human, but the human designs an algorithm to create these tests in an automated manner. The capabilities of these automated tests to find bugs can be enhanced by having means to detect in runtime when security properties may be violated.

Manual testing

Exhaustive: cover all inputs
Not feasible due to massive state space

Functional: cover all requirements
Depends on specification

Random: automate test generation
Incomplete (what about that hard check?)

Structural: cover all code
Works for unit testing

When manual tests are defined by humans, they can be guided by the following principles:

- One can try to cover all inputs. Can be infinite, and even when finite will typically be unfeasible as the number of inputs grows exponentially with the number of new variables and branch conditions

- One can try to cover all requirements, extracting those from the specification. This requires that the specification is available, and that all requirements can be well identified from the spec itself.

- One can perform random checks by generating them automatically. It is difficult that completely random checks achieve good code coverage, as they will never pass hard checks in which the program checks for a particular value. For instance:

```
if (a = sha256("Hello World"))
```

- Finally we can try to cover all code, but this only works for small code bases (e.g., for unit tests)

Automated testing

Static analysis

Analyze the program without executing it
Imprecision by lack of runtime information, e.g. aliasing

Symbolic analysis

Execute the program symbolically
Keeping track of branch conditions
Not scalable

Dynamic analysis (e.g., fuzzing)

Inspect the program by executing it
Challenging to cover all paths

On the other hand, one can design algorithms to define the tests to run.

These algorithms can be based on one of these three approaches:

Static analysis: this type of analysis analyzes the code *without executing it*. Static analysis can find basic errors. This method is limited by its incapability to know what the values of variables are going to be in runtime. It can also not catch race conditions that happen when an address is pointed by more than one variable (this phenomenon is also called “aliasing”).

Symbolic analysis: this analysis computes an approximation of what the program actually does by constructing formulas representing the program state at various points. It is called "symbolic" because the approximation relies on representing the program, and its different branches, as logic formulas.
Symbolic execution identifies decision points (e.g., if statements) and associates them with logical variables that can be met or not.
This approach is extremely effective, as it can consider all paths, but it requires to keep an enormous amount of state (remember all options for all decision points). It cannot scale to large pieces of code.
Symbolic execution will also perform poorly when the program includes calls to components that are not under the control of the program itself (e.g., calls to the

system); or when memory regions are accessed using different names.

Dynamic analysis: this analysis consists on executing the code with diverse inputs. The main challenge is to cover all the paths (control- and data-flows).

Coverage: testing needs a metric

Why use Coverage?

Intuition: A software flaw is only detected if the flawed statement is executed!
Effectiveness of test suite therefore depends on how many statements are executed.

Statement coverage

how many statements (e.g., an assignment, a comparison, etc.) in the program have been executed

Branch coverage

how many branches among all possible paths have been executed

To measure how complete a set of tests is we use the concept of **coverage** with aims at quantifying how many statements of the program are executed by the tests.

Coverage can be measured with respect to different elements:

Statement coverage: measures how many statements (e.g., an assignment, a comparison, etc.) in the program have been executed.
Statement coverage does not mean full coverage. All statements may be executed at least once, but if the values of the variables are limited, some errors may not be found.

Branch coverage: measures how many branches among all possible paths have been executed. In other words, for each branch in the program (e.g., if statements, loops), how many branches have been executed at least once during testing.
Branch coverage is neither complete. All branches may be executed, but that does not mean we have tested all the values of the variables.

Coverage: testing needs a metric

```
int func(int elem, int *inp, int len) {
  int ret = -1;
  for (int i = 0; i <= len; ++i) {
    if (inp[i] == elem) { ret = i; break; }
  }
  return ret;
}
```

Test input: elem = 2, inp = [1, 2], len = 2 results in full ***statement coverage***.

Loop is never executed to termination, where the out of bounds access happens.
Statement coverage does not imply ***full*** coverage.

Current practice is ***branch coverage***

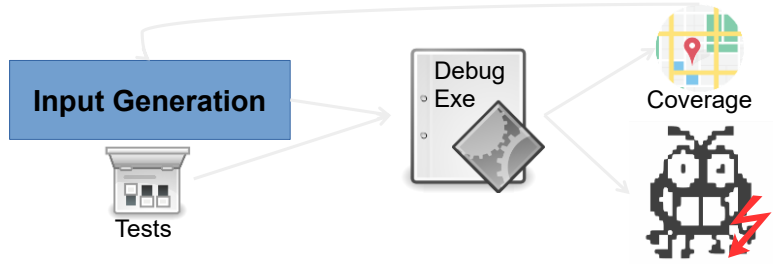
In this example, for inputs:
elem = 2
inp = [1, 2]
len = 2
We would run **all** the statements. So statement coverage is complete.

However, the loop would never finish, so we would not find the bug that happens when i>length(inp)

Fuzzing

A random testing technique that mutates input to improve test coverage

State-of-art fuzzers use coverage as feedback to mutate the inputs



Fuzzing input generation

Dumb Fuzzing is unaware of the input structure; randomly mutates input

Generation-based fuzzing has a model that describes inputs; input generation produces new input seeds in each round

Mutation-based fuzzing leverages a set of valid seed inputs; input generation modifies inputs based on feedback from previous rounds

Mutations can be informed by structure *white-box, grey-box, black-box.*

There are different criteria to create the list of inputs to be tested:

Dumb: generate the list of inputs at random, i.e., do not consider the relation between inputs nor follow a model

Generation-based: use a model that constraints how the inputs are generated. generates a new input each round according to the model.

Mutation-based: instead of generating new inputs according to a model, this approach modifies the inputs according to some rules considering previous inputs and what the results obtained by those inputs.

If the algorithm has information about the program, this information can be used to design new inputs in a more effective way.

White box: In a white-box model, the fuzzer has full knowledge of the code. This can be used to chose optimal mutations (e.g., find inputs that unblock hard checks)
Grey box: does not know all the instructions, but can use results from previous rounds to infer information about the program and optimize the choices of inputs.
Black box: generates the inputs without knowledge of the program.

Sanitization

Test cases detect bugs through
- Assertions
  - assert(var!=0x23 && "illegal value");
- Segmentation faults
- Division by zero traps
- Uncaught exceptions
- Mitigations triggering termination

How can we increase bug detection chances?
- *Sanitizers* enforce some policy, detect bugs earlier and increase effectiveness of testing.

Tests and fuzzing can find errors associated to operations, but only when the errors happen.

Sanitizers try to catch these errors before they happen so that testing can be faster.

Address Sanitizer

AddressSanitizer (ASan) detects memory errors. It places red zones around objects and checks those objects on trigger events.

The tool can detect the following types of bugs:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return (configurable)
- Use-after-scope (configurable)
- Double-free, invalid free
- Memory leaks (experimental)

Slowdown introduced by AddressSanitizer is 2x.

Every time a variable is defined **AddressSanitizer (Asan)** marks the memory locations around this variable as “red zones”, i.e., zones that the program should not catch. It does the same with parts of the memory that are freed and the program should not touch again.

These records are stored in a “shadow memory” that is checked on runtime. If at any point ASan detects that a red zone will be accessed it raises an alarm. As such, it detects the error before it happens.

It is a quite light countermeasure, only doubles execution time.

Undefined behavior Sanitizer

UndefinedBehaviorSanitizer (UBSan) detects undefined behavior. It instruments code to trap on typical undefined behavior in C/C++ programs.

Detectable errors are:

- Unsigned/misaligned pointers
- Signed integer overflow
- Conversion between floating point types leading to overflow
- Illegal use of NULL pointers
- Illegal pointer arithmetic
- ...

Slowdown depends on the amount and frequency of checks. This is the only sanitizer that can be used in production. For production use, a special minimal runtime library is used with minimal attack surface.

The idea behind the **UndefinedBehaviorSanitizer (UBSan)** is to detect problems that happen as a consequence of a undefined behavior (typically source of danger).
For instance, when two pointers read/write from the same location and they are not synchronized, and executing an instruction of this location results on undefined behavior.

UBSan records the location of pointers and checks that the memory they point to is in the expected state when they access it.

If you want to learn more about sanitizers

- AddressSanitizer: https://clang.llvm.org/docs/AddressSanitizer.html
- LeakSanitizer: https://clang.llvm.org/docs/LeakSanitizer.html
- MemorySanitizer: https://clang.llvm.org/docs/MemorySanitizer.html
- UndefinedBehaviorSanitizer: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
- ThreadSanitizer: https://clang.llvm.org/docs/ThreadSanitizer.html

Software Security: Summary

Reminders (that we never FULLY understand)

- Code is data (Endless source of joys and nightmares since 1947)
- Data (input) from the user can become code … DaNgEr …Boooom
- Abstraction gap between C and assembly (e.g. RISC-V) is nontrivial.
  - Gremlins are hiding in the details

Two approaches: mitigations and testing for catching gremlins

Mitigations stop unknown vulnerabilities

- Make exploitation harder, not impossible

Testing discovers bugs during development

- Automatically generate test cases through fuzzing
- Make bug detection more likely through sanitization