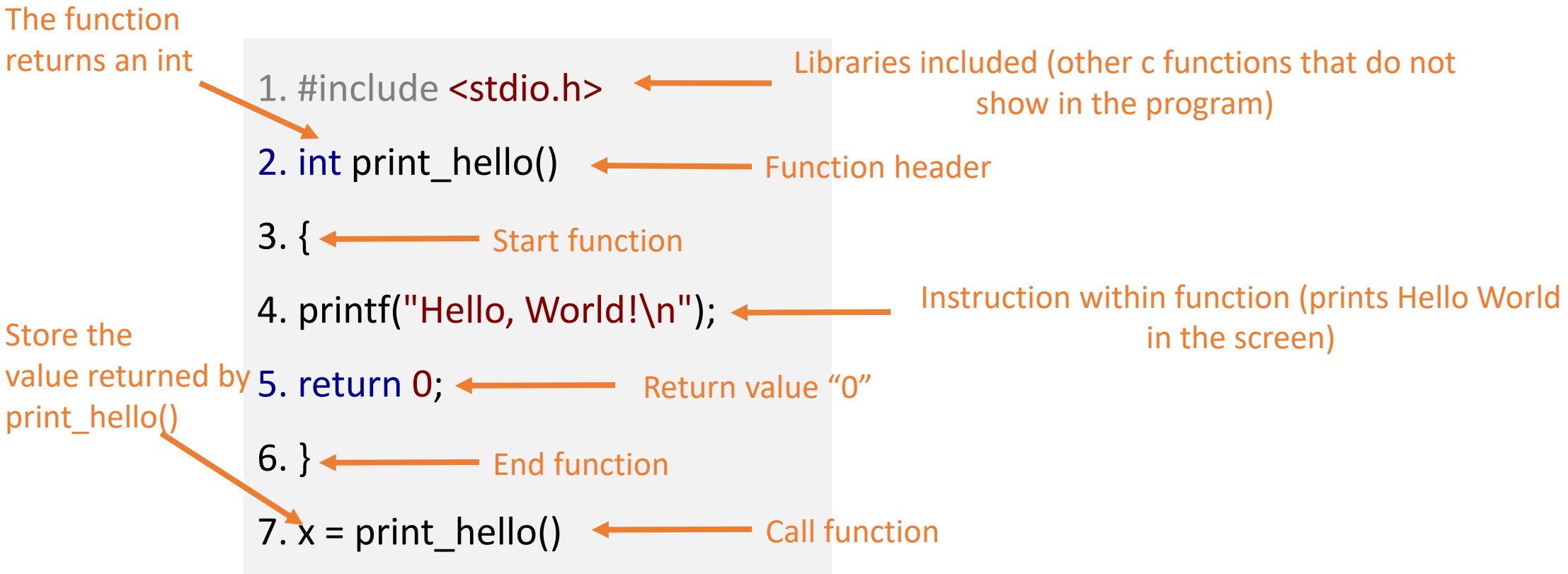
Midterm

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



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)

* Indicates a *pointer*: a pointer is a special variable that stores addresses rather than values

& Returns the address of a variable

```
1. int* pc, c;
2. c = 5;
3. pc = &c;
4. printf("%d", *pc);
```

Returns the content of in the address pointed by a pointer
(in this case, the content of the address pointed by pc is the address of the variable c)

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

Layout of a C program

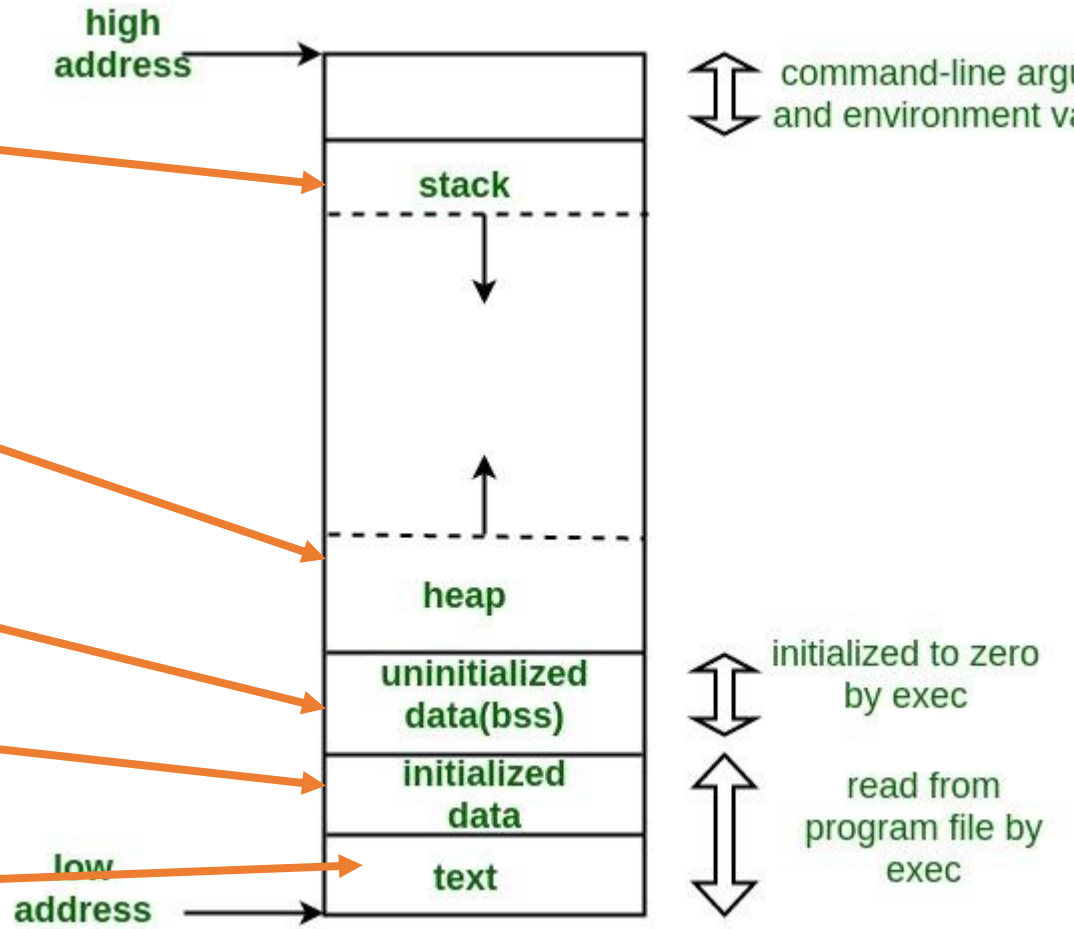
LIFO structure
function variables, along with information that is saved each time a function is called

dynamic memory allocation
(malloc, calloc,…)

global variables and static variables that are initialized to zero or do not have explicit initialization in source code

global variables and static variables that are initialized by the programmer

Executable instructions (code). It is under the heap and the stack with the goal of avoiding that it gets rewritten



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).

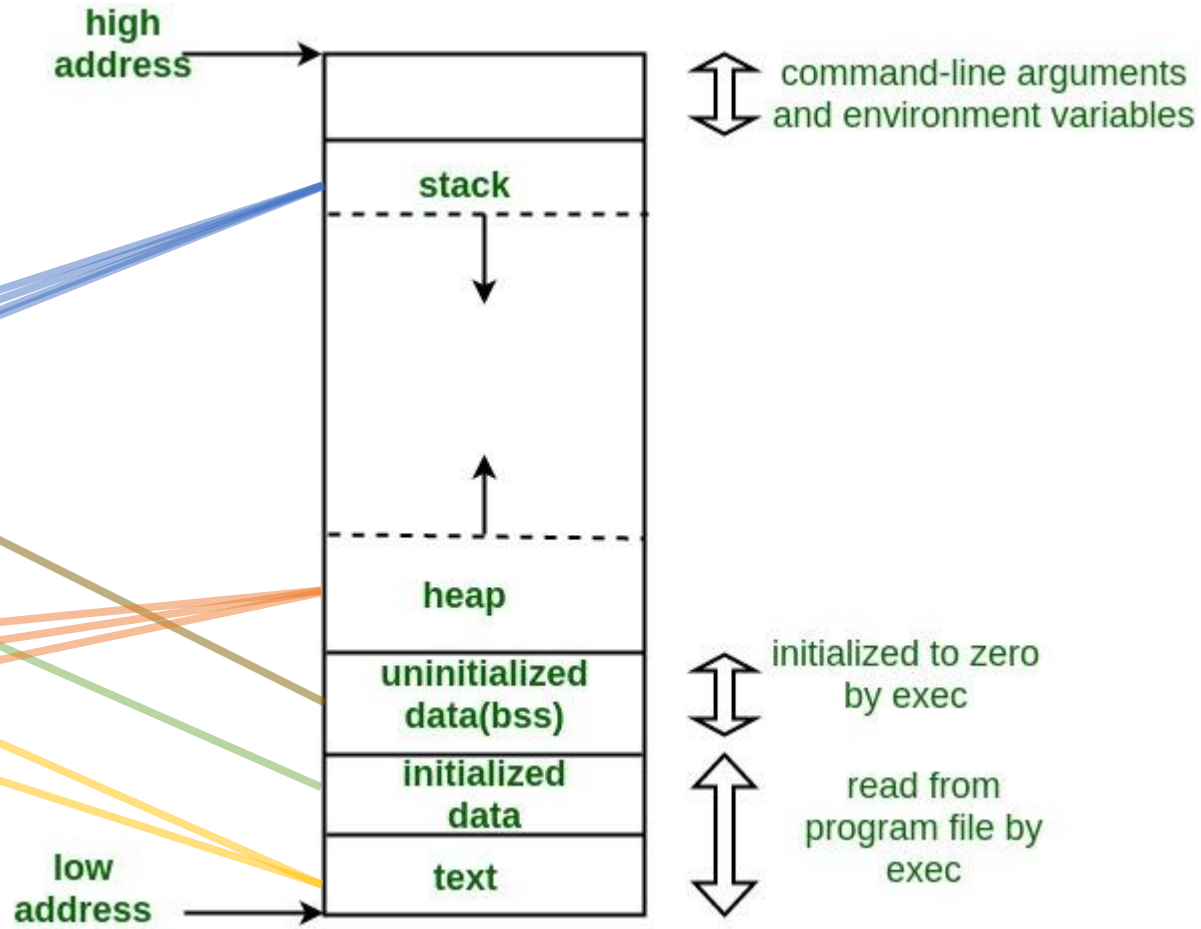
2. At the end of process_data, we call g
   Can we still access the address stored in variable buffer within function g?
   Can we access the data at D within function g?

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

C language 101: concepts for the lecture

Layout of a C program

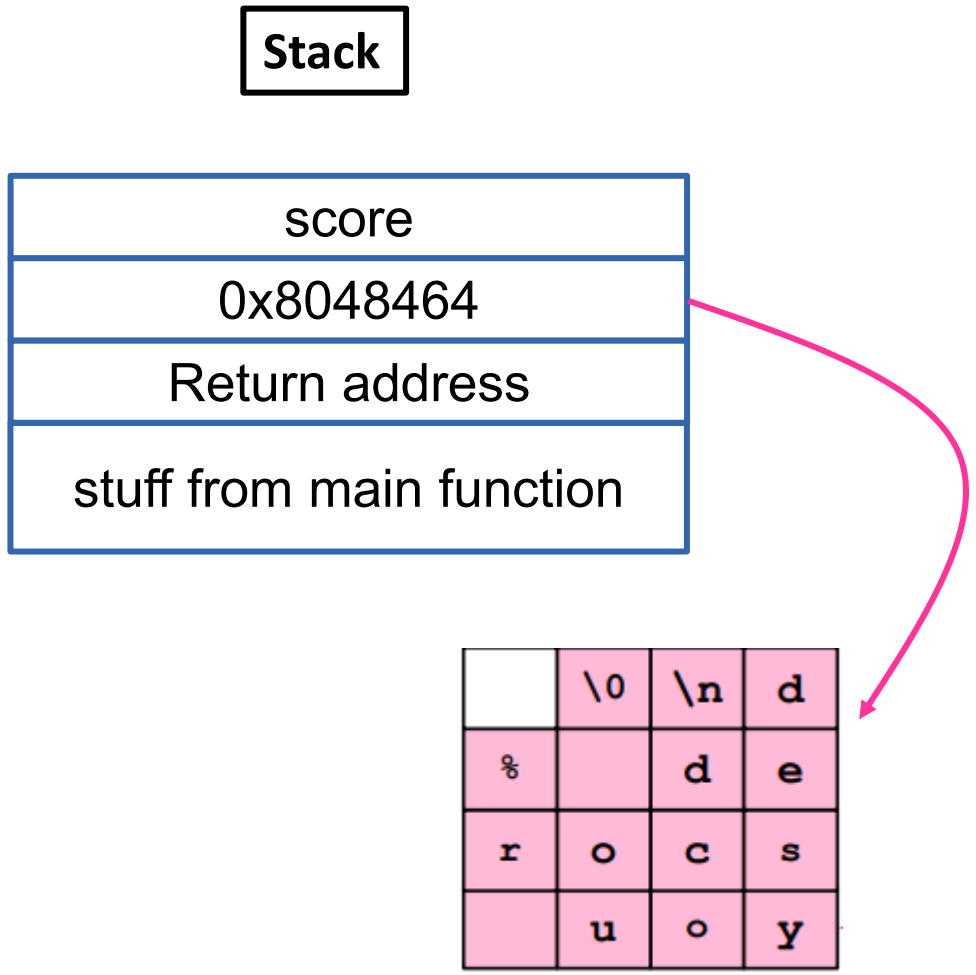
```
char big_array[100];
char huge_array[1000];
int global = 0;

int useless() { return0; }

int main() {
    void *p1, *p2, *p3;
    int local = 0;
    p1 = malloc(28);
    p2 = malloc(8);
    p3 = malloc(32);
}
```



C language 101: concepts for the lecture

Calling a function

```
int __printf (const char *format, ...) {
    Code to print things;
}

int main {
 /* code doing stuff */
printf("You scored %d\n", score)
 /* code doing stuff */
}
```



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?)

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
}
```

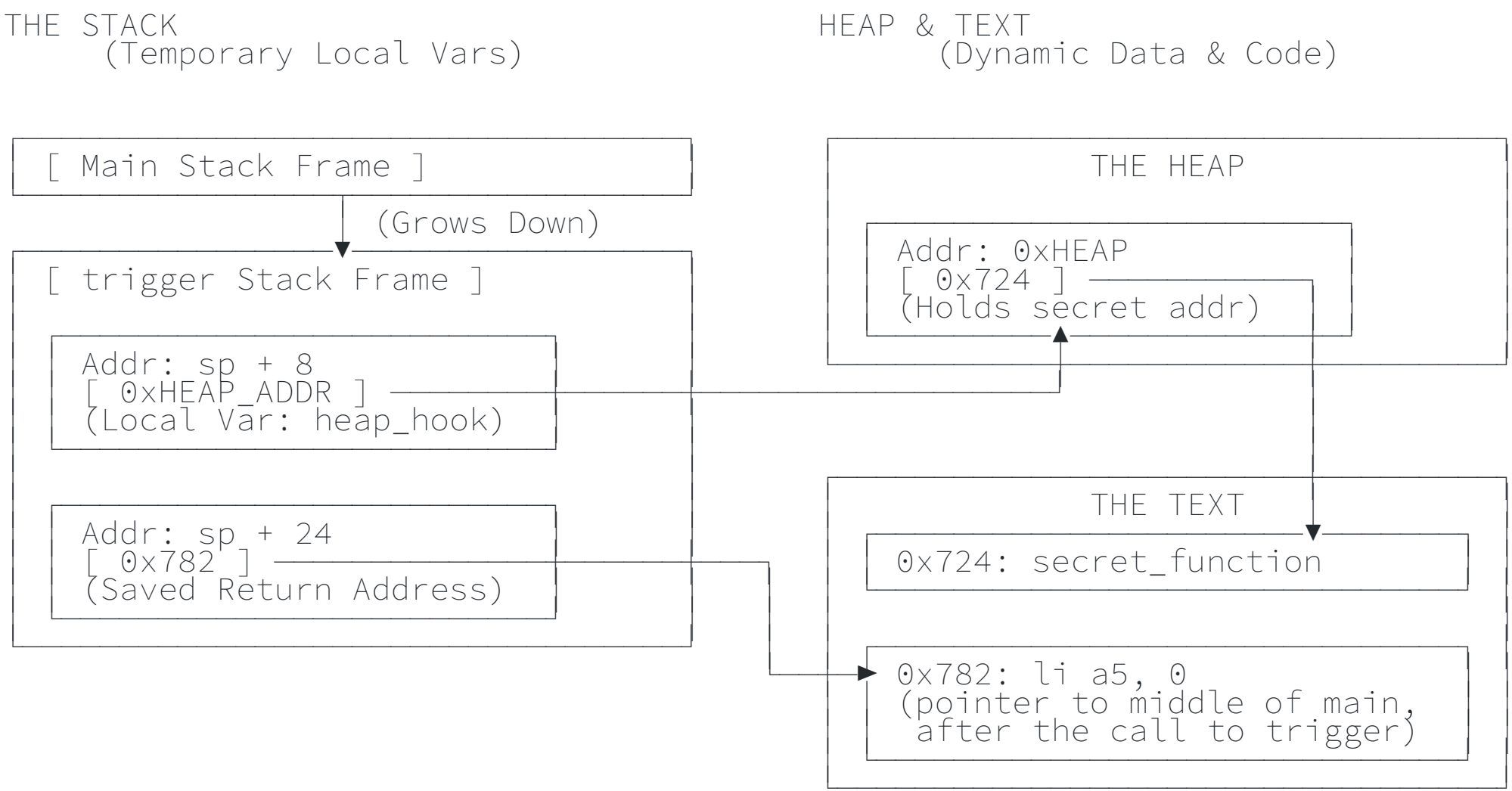
Q2 The ABI/Stack Frame: When `main` calls `trigger()`, the system will push a value onto the stack (RISC-V/x86-64. What specifically is that value?

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
```

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
```


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…

Traveler Information

Traveler 1 - Adults (age 18 to 64)

To comply with the TSA Secure Flight program, the traveler information listed here must exactly match the information on the government-issued photo ID that the traveler presents at the airport.

Title (optional): Dr.

First Name: Alice

Middle Name:

Last Name: Smith

Travelers are required to enter a middle name/initial if one is listed on their government-issued photo ID.

Gender: Female

Date of Birth: 01/24/93

Some younger travelers are not required to present an ID when traveling within the U.S. Learn more

+ **Known Traveler Number/Pass ID (optional):**

+ **Redress Number (optional):**

Seat Request:
(•) No Preference ( ) Aisle ( ) Window

Traveler Information

Traveler 1 - Adults (age 18 to 64)

To comply with the TSA Secure Flight program, the traveler information listed here must exactly match the information on the government-issued photo ID that the traveler presents at the airport.

Title (optional): Dr.

First Name: Alice

Middle Name:

Last Name: Smithhhhhhhhhhhhh

Travelers are required to enter a middle name/initial if one is listed on their government-issued photo ID.

Gender: Female

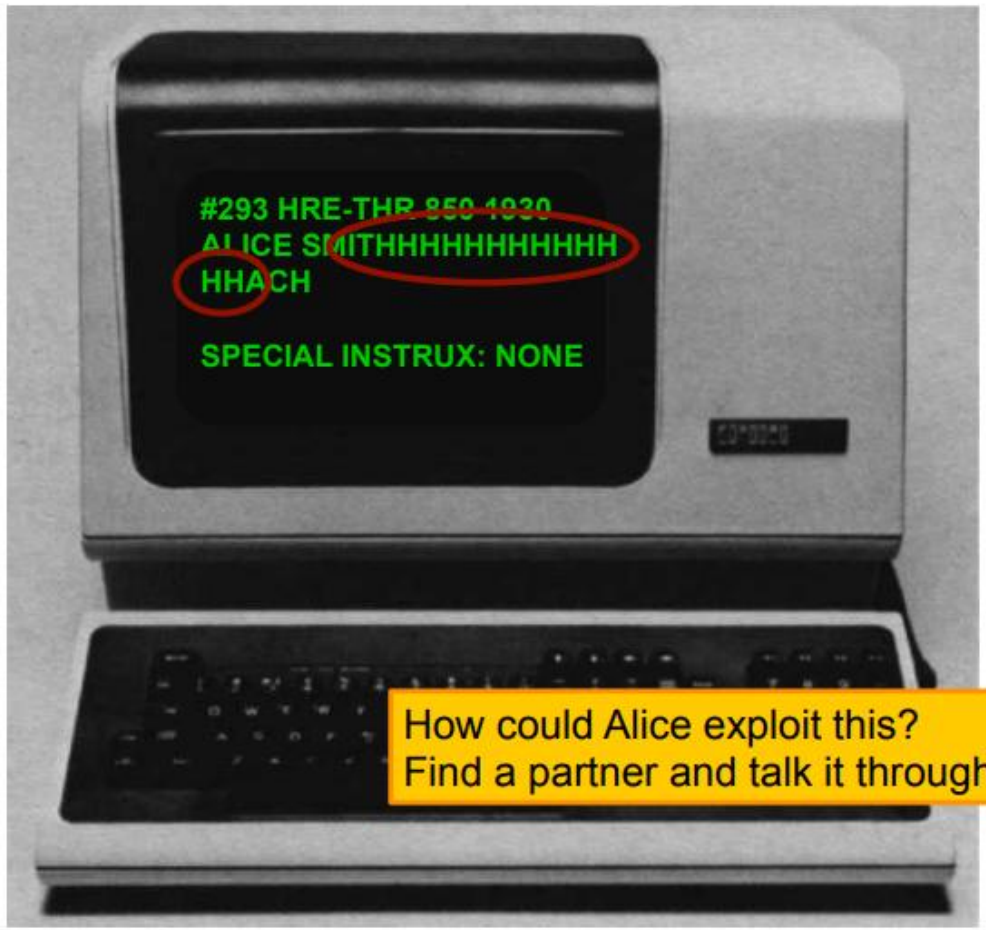
Date of Birth: 01/24/93

Some younger travelers are not required to present an ID when traveling within the U.S. Learn more

+ **Known Traveler Number/Pass ID (optional):**

+ **Redress Number (optional):**

Seat Request:
(•) No Preference ( ) Aisle ( ) Window



Traveler Information

Traveler 1 - Adults (age 18 to 64)

To comply with the TSA Secure Flight program, the traveler information listed here must exactly match the information on the government-issued photo ID that the traveler presents at the airport.

Title (optional): Dr.

First Name: Alice

Middle Name:

Last Name: Smith First

Travelers are required to enter a middle name/initial if one is listed on their government-issued photo ID.

Gender: Female

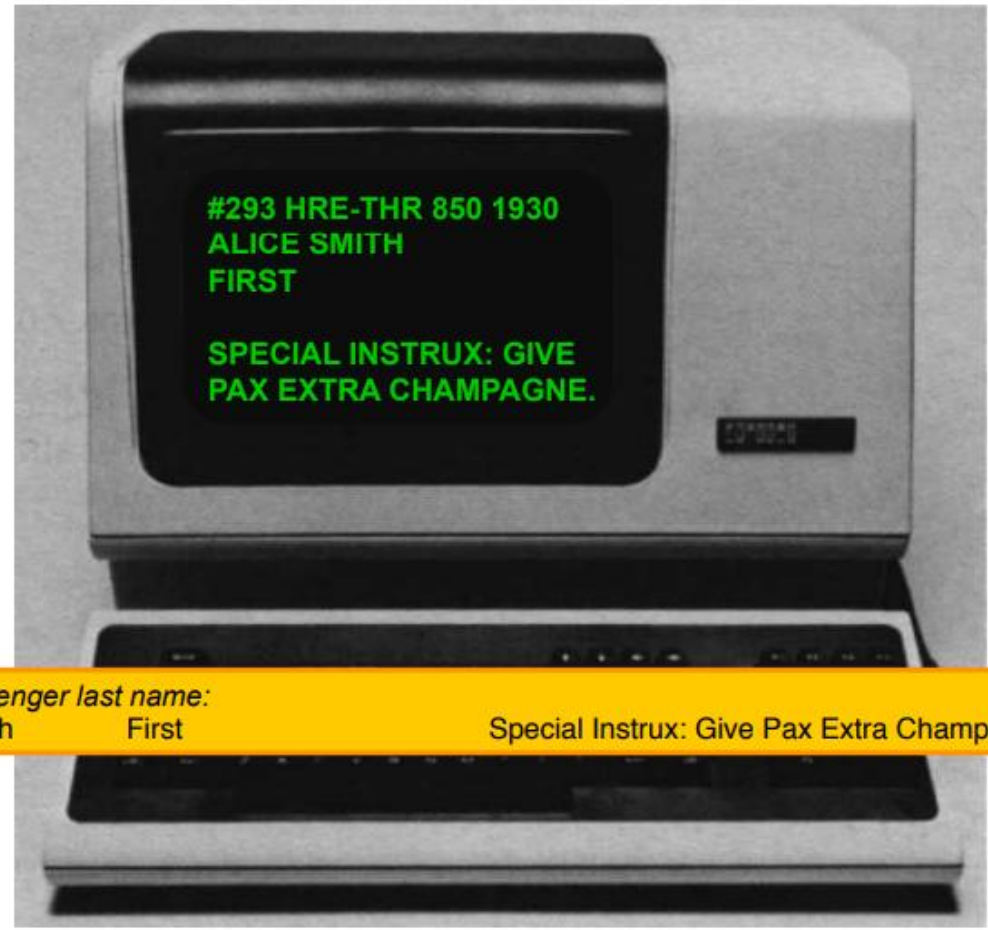
Date of Birth: 01/24/93

Some younger travelers are not required to present an ID when traveling within the U.S. Learn more

+ **Known Traveler Number/Pass ID (optional):**

+ **Redress Number (optional):**

Seat Request:
(•) No Preference ( ) Aisle ( ) Window



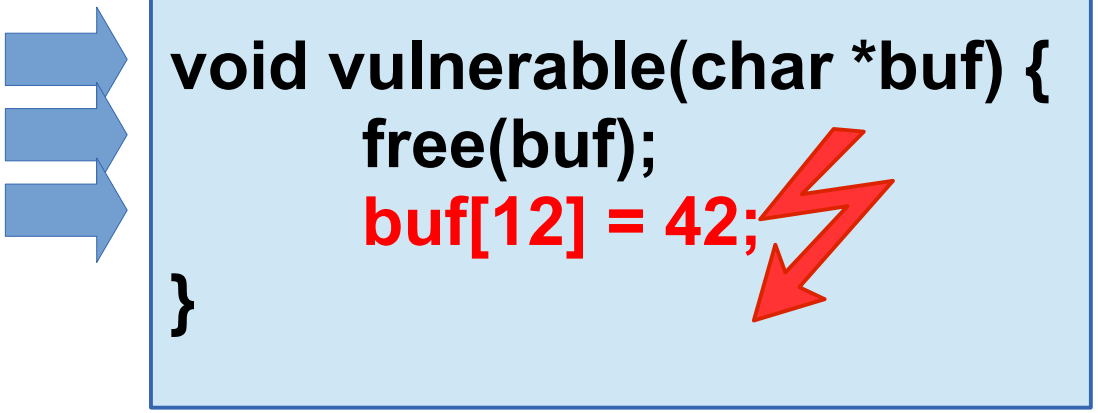
Passenger last name:
"Smith First Special Instrux: Give Pax Extra Champagne."

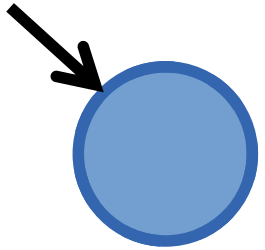
Memory corruption

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

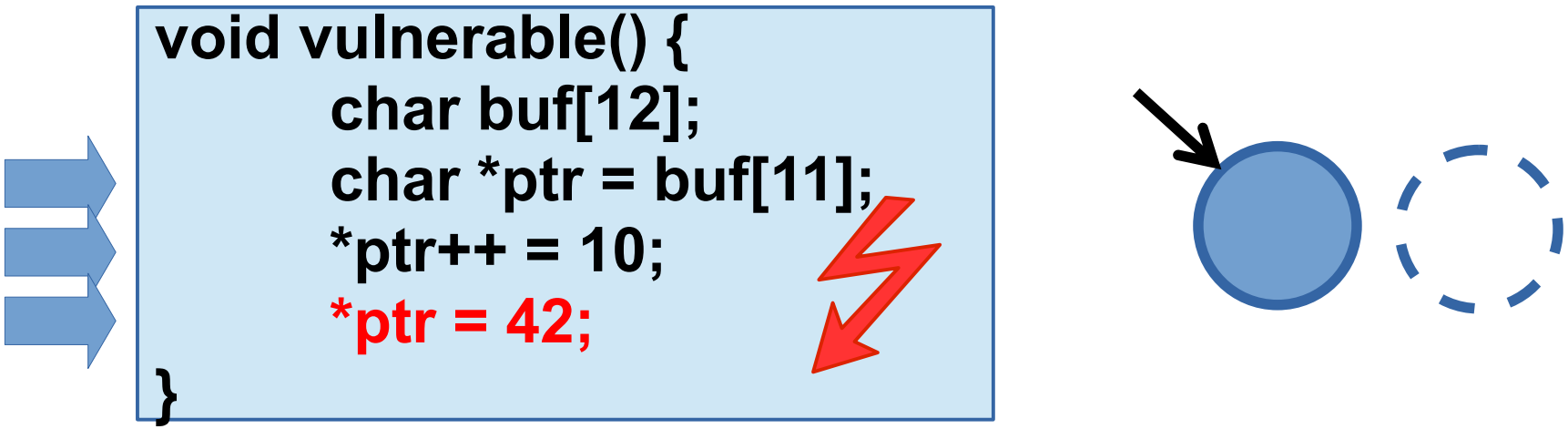
```
void vulnerable(int user1, int *array) {
        array[user1] = 42;
}
```

Memory safety: temporal error





Memory safety: spatial error



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`

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;
}
```

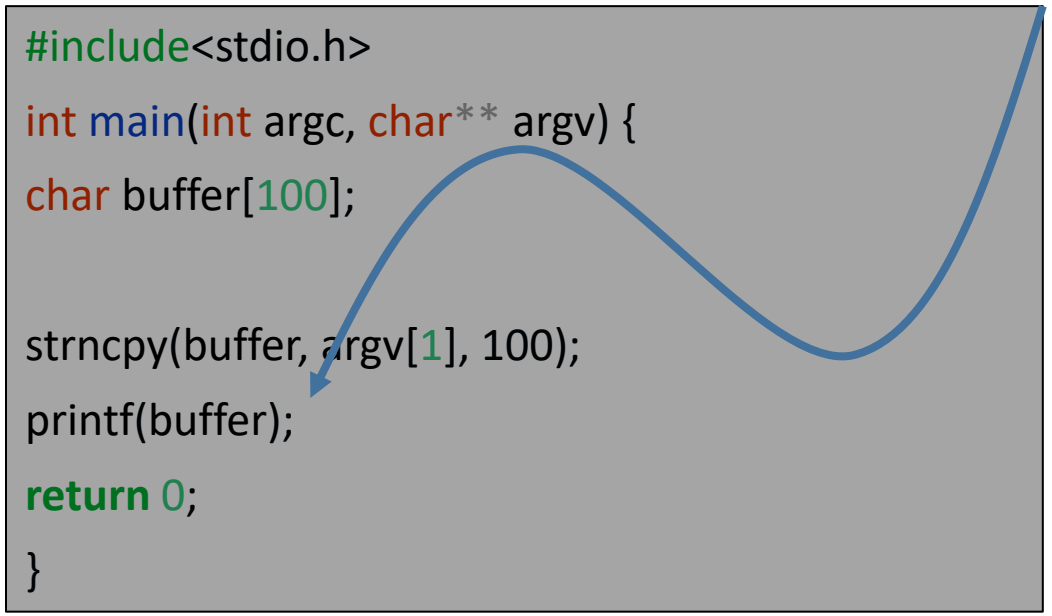
Question 1: What does `./myProgram` do? What does `./myProgram Hello` do?

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

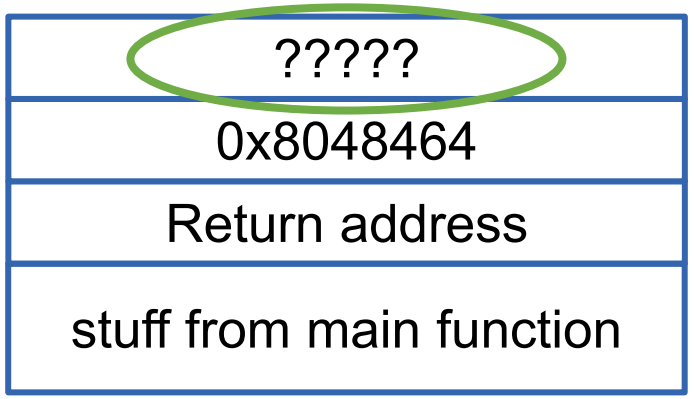
Uncontrolled Format String *(CWE-134)*

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;
}
```



4 bytes from the stack!



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

```
#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

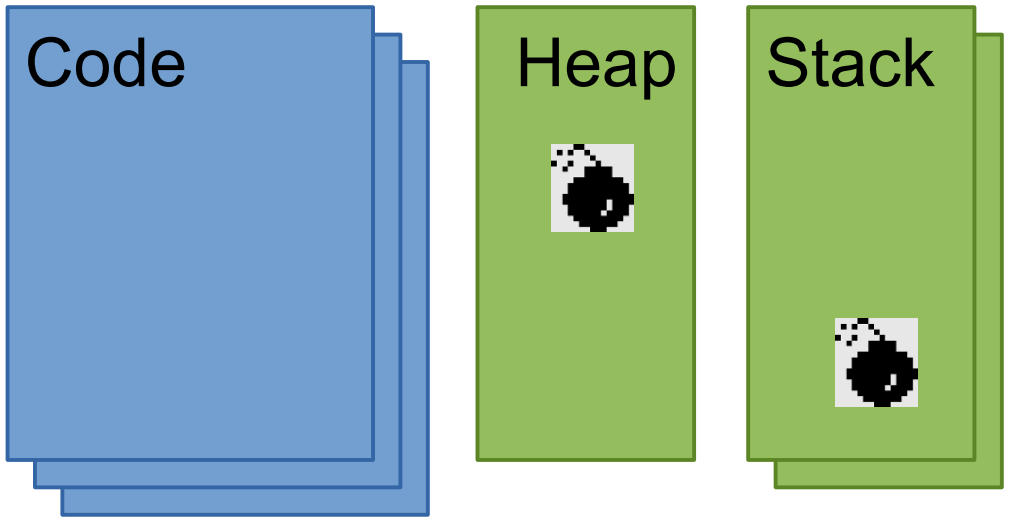
The programmer should decide the format of the string. That ensures that no extra argument, read or write, can be used.

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

Attack scenario: code injection

Force memory corruption to set up attack

Redirect control-flow to injected code



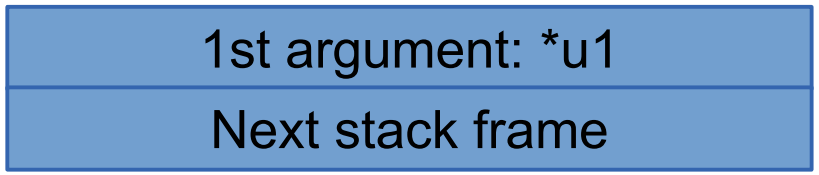
Code injection attack

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

Next stack frame

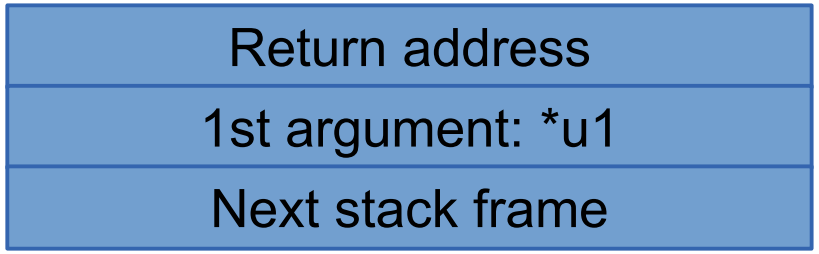
Code injection attack

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



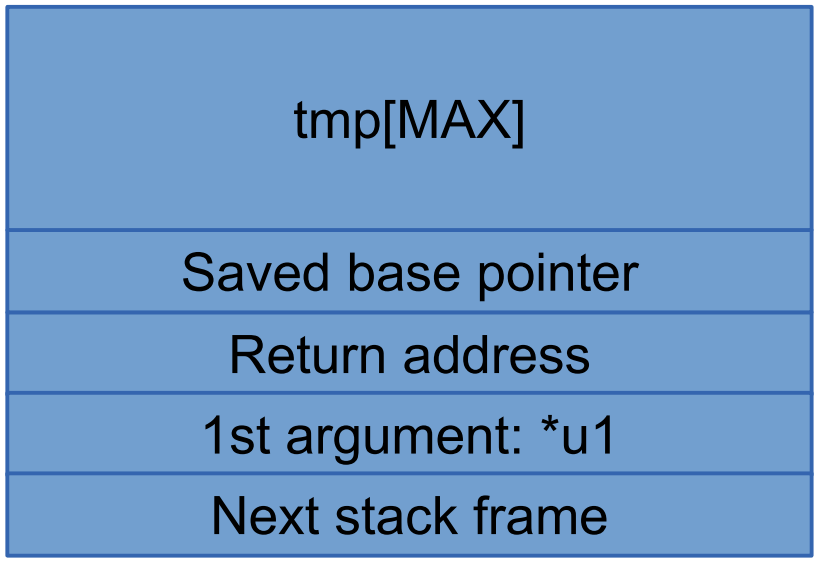
Code injection attack

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



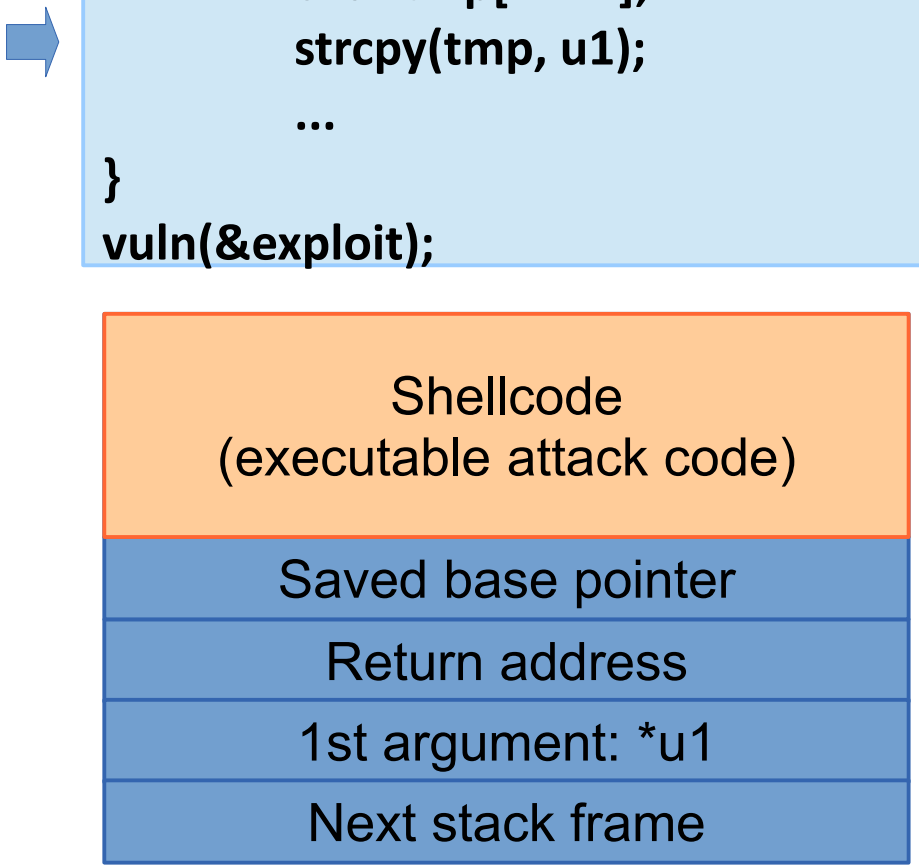
Code injection attack

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



Code injection attack

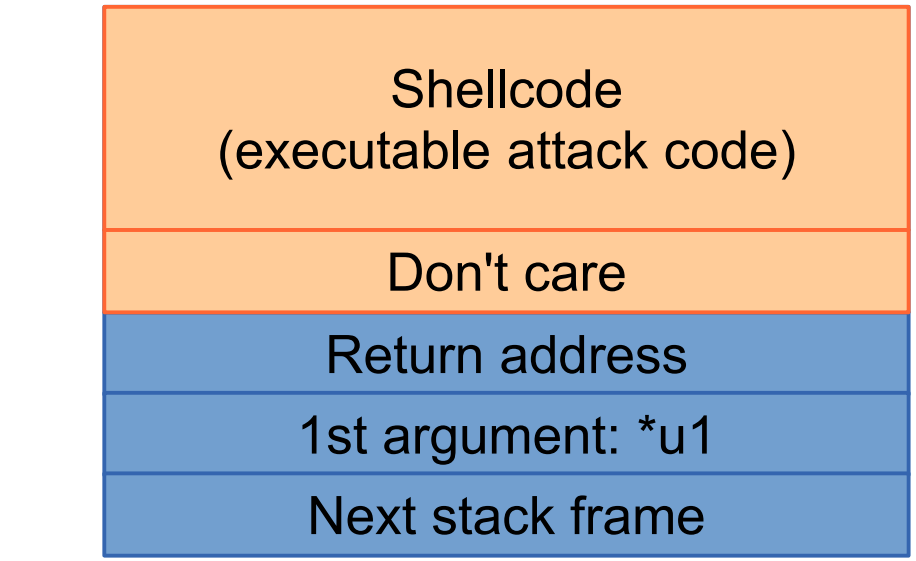
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```



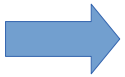
Code injection attack

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

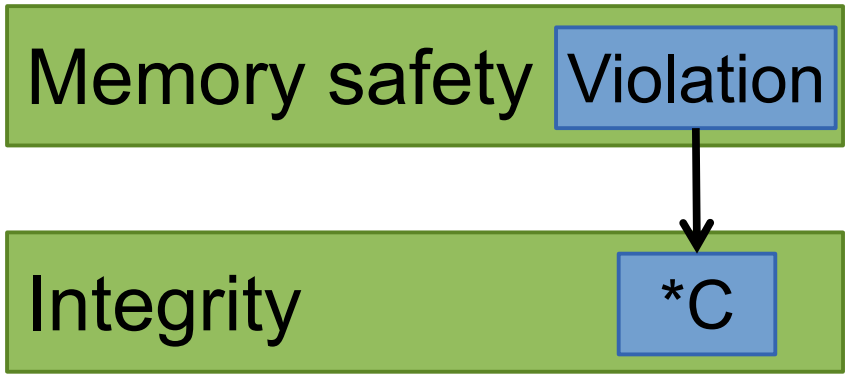
Memory safety Violation

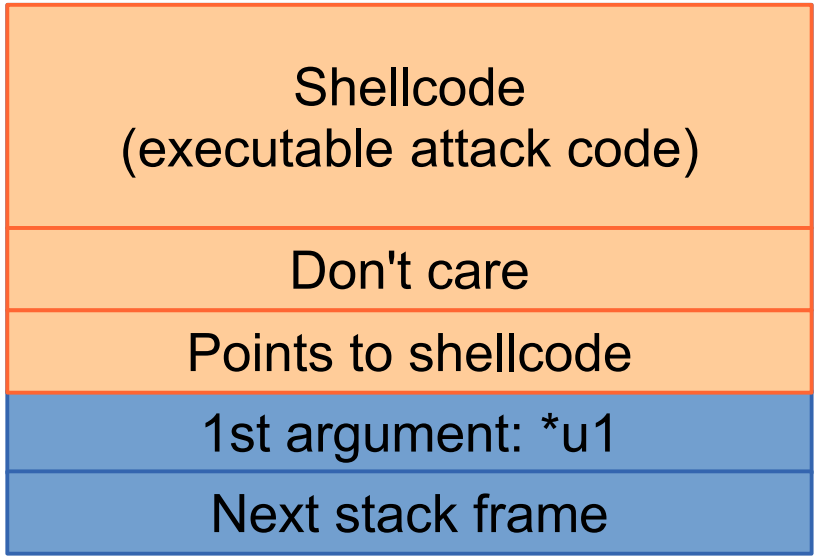


Code injection attack



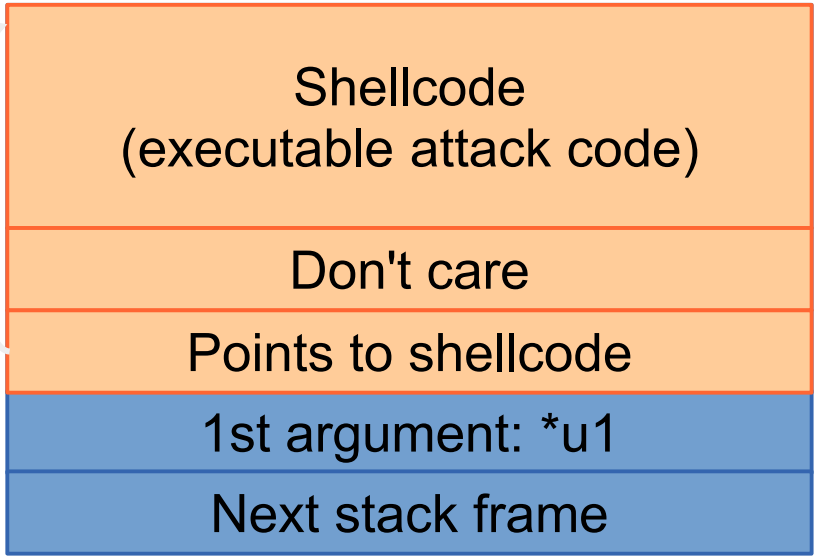
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```

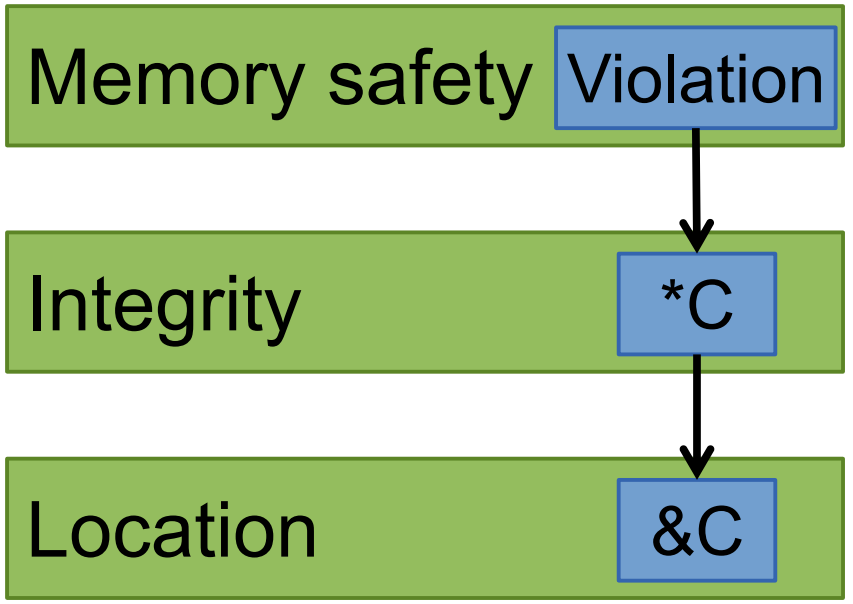




Code injection attack

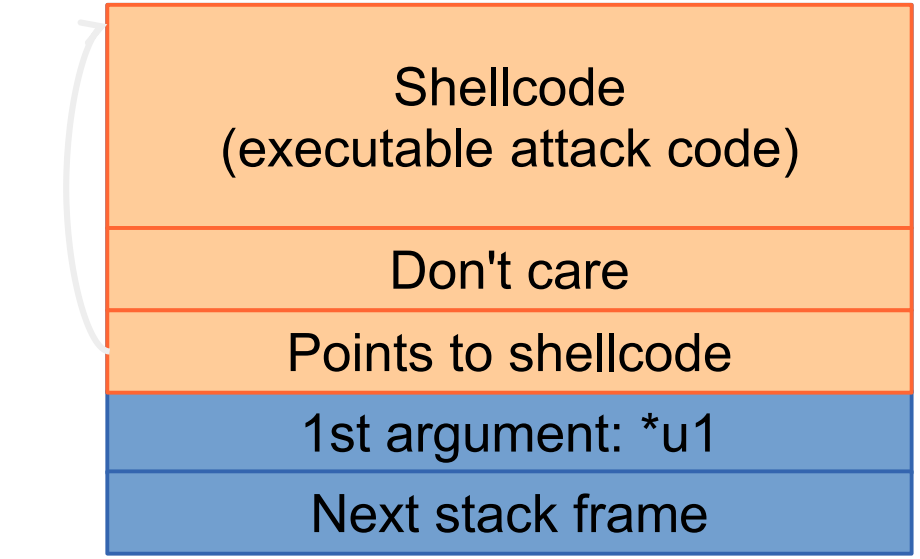
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```

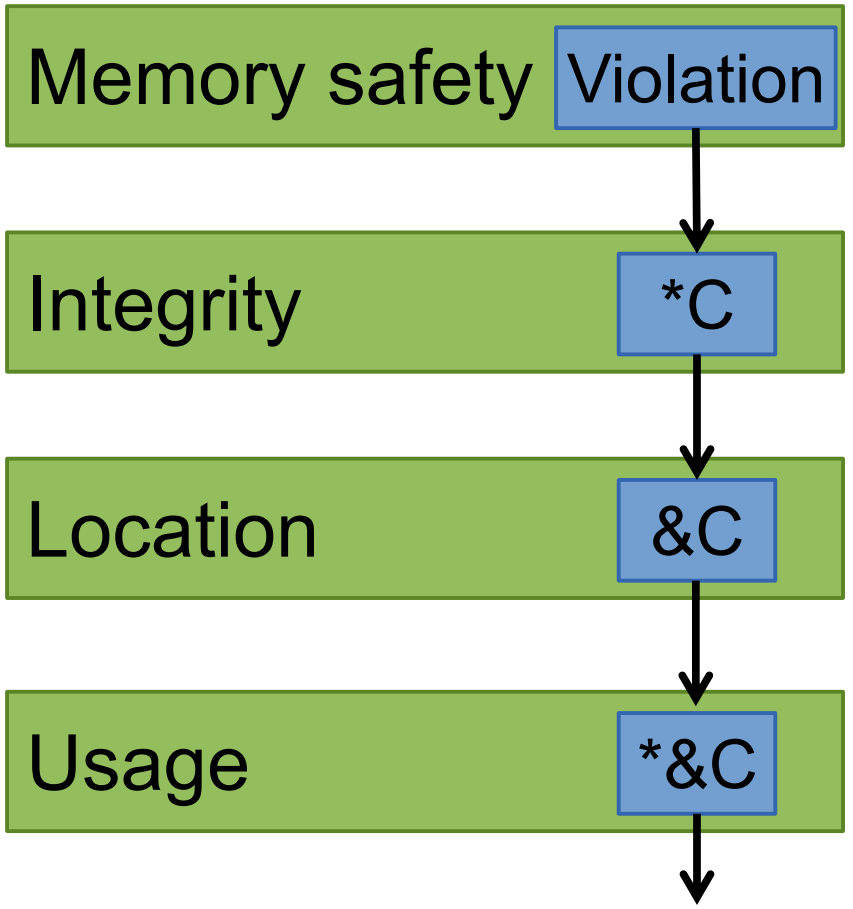




Code injection attack

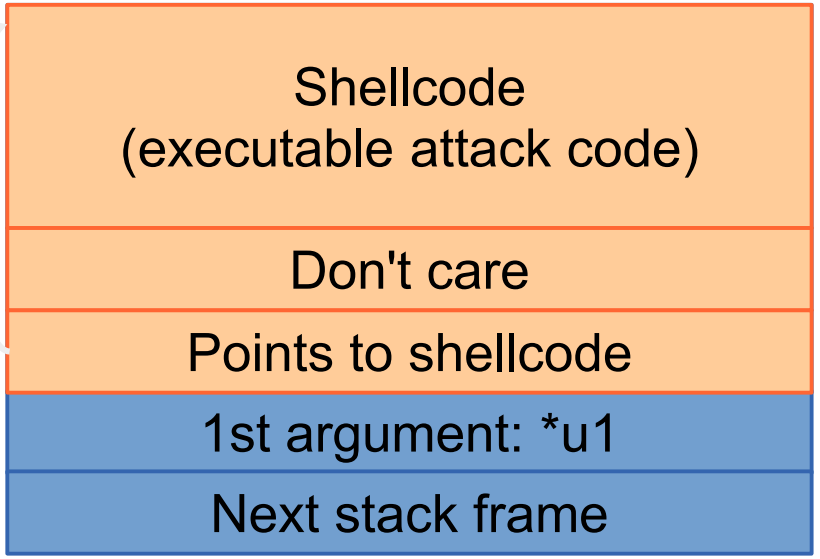
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```

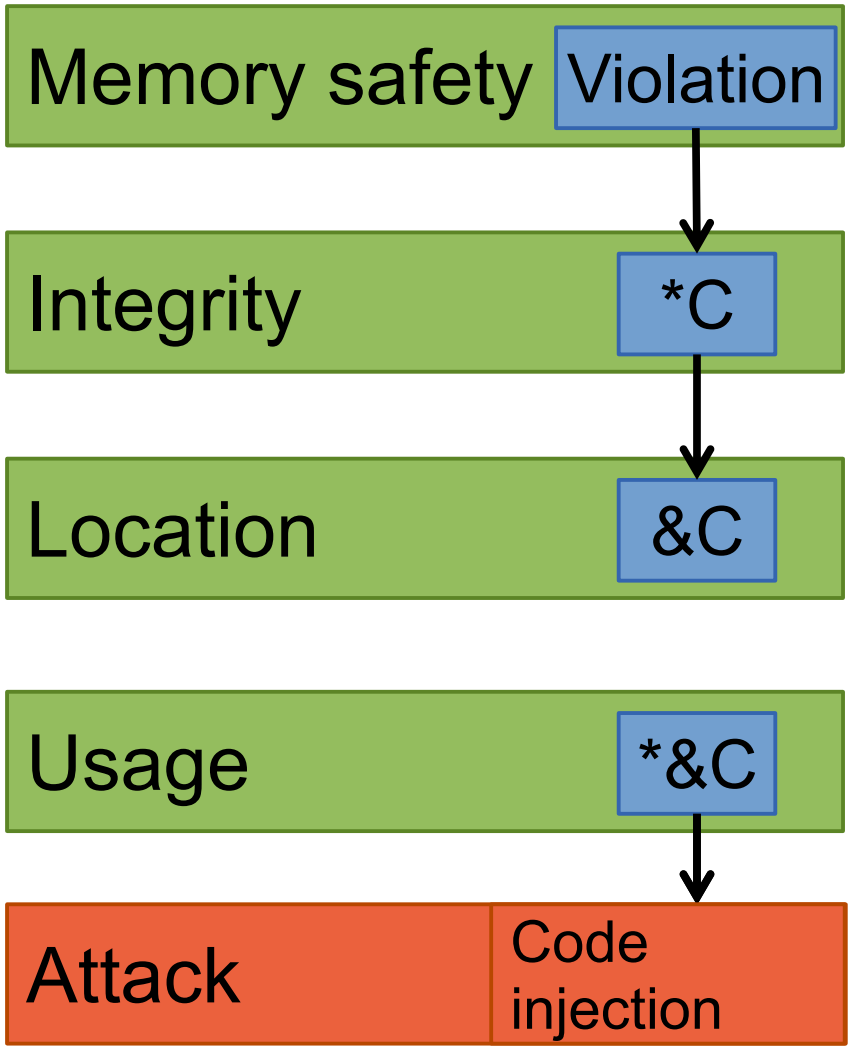




Code injection attack

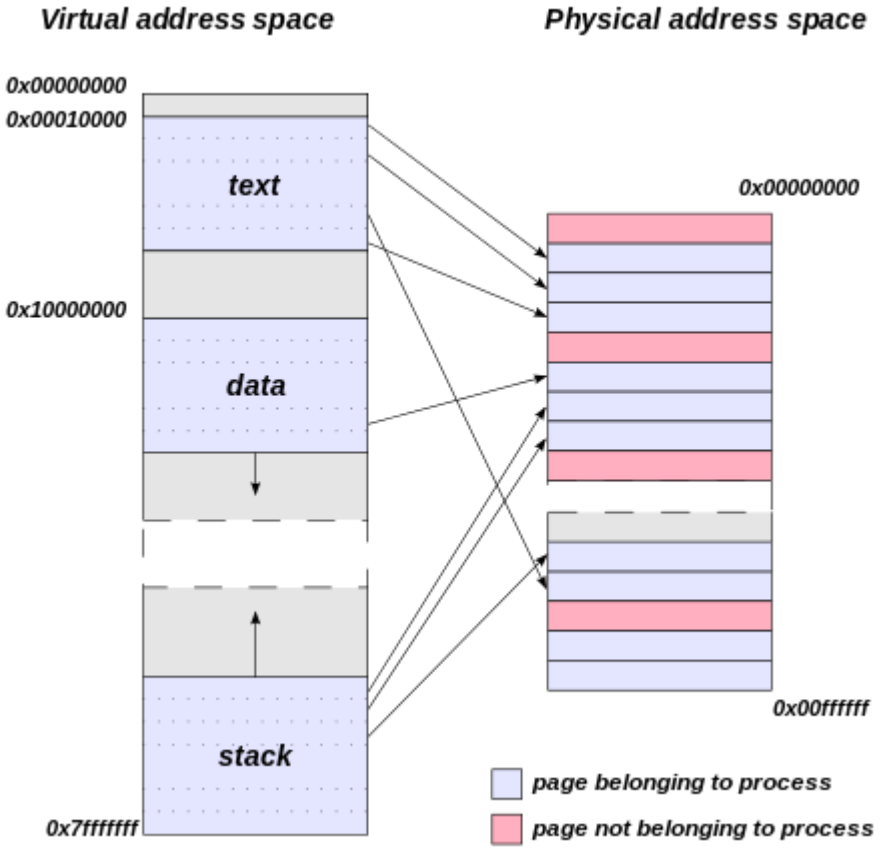
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```





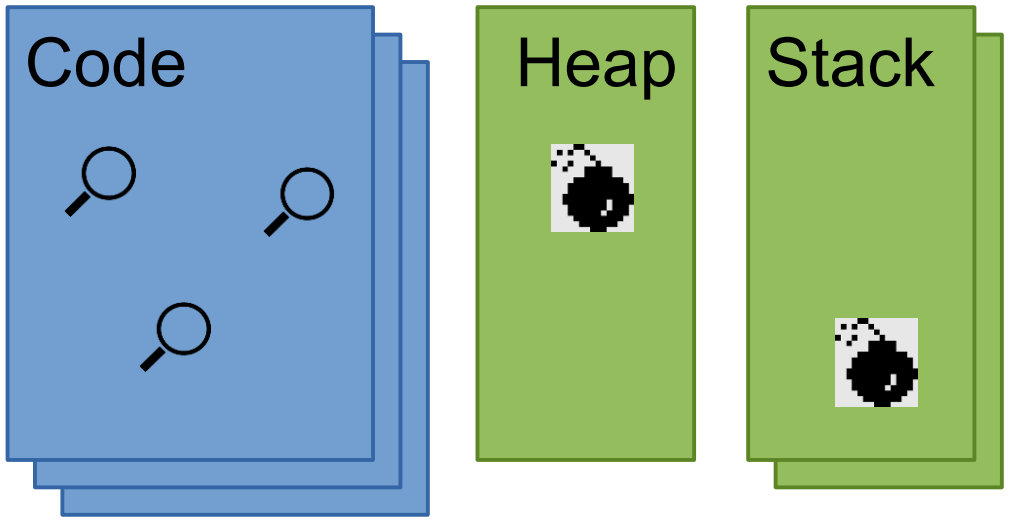
Data Execution Prevention

- 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



Attack scenario: code reuse

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



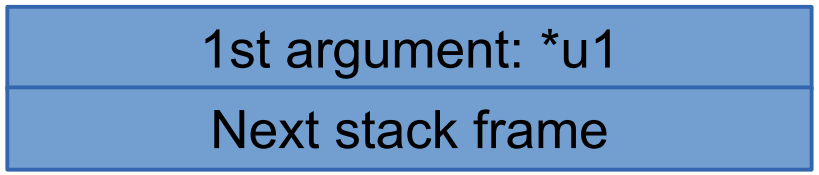
Control-flow hijack attack

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

Next stack frame

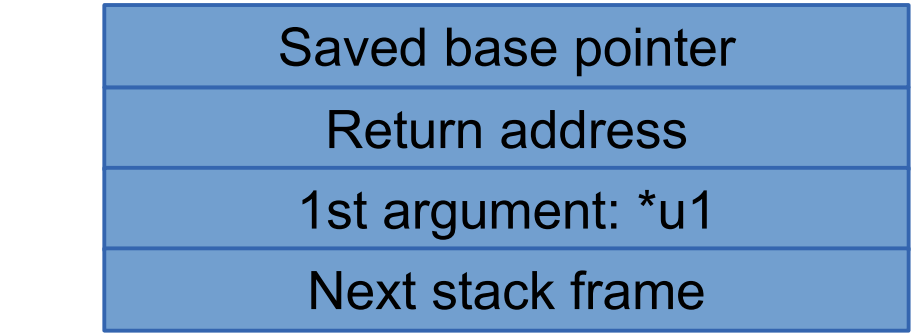
Control-flow hijack attack

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



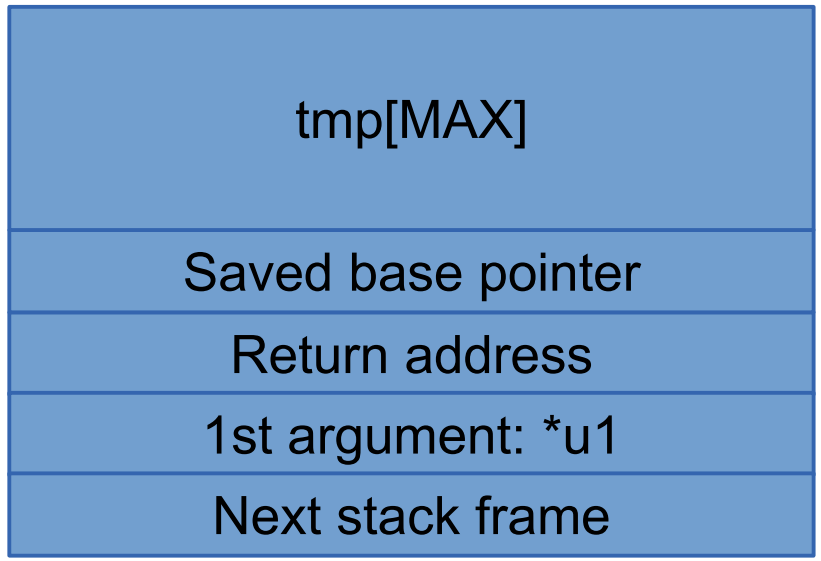
Control-flow hijack attack

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



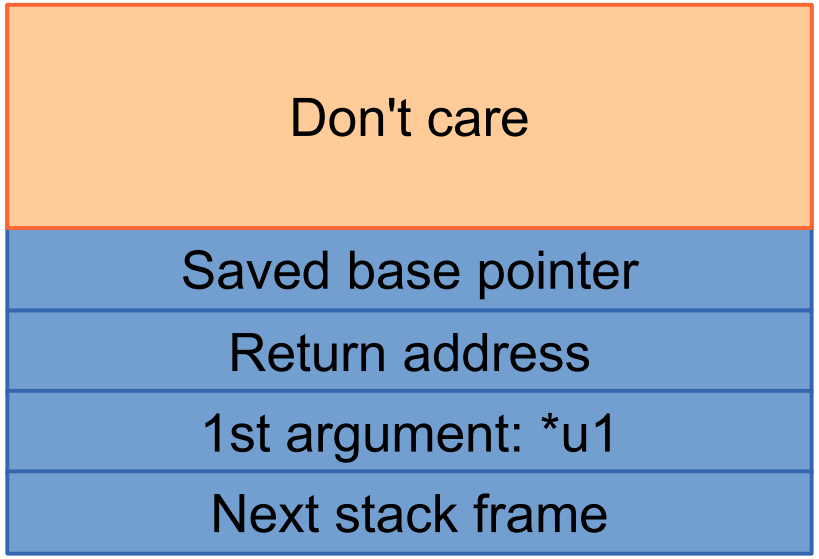
Control-flow hijack attack

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



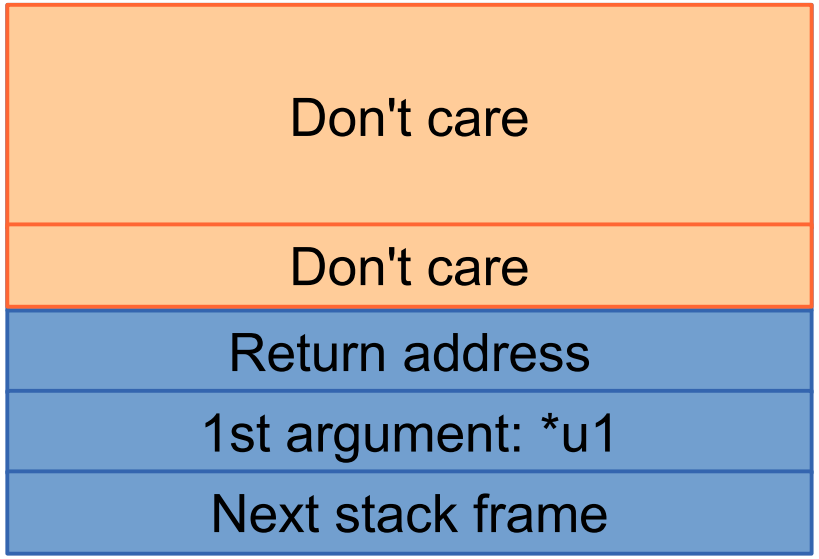
Control-flow hijack attack

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



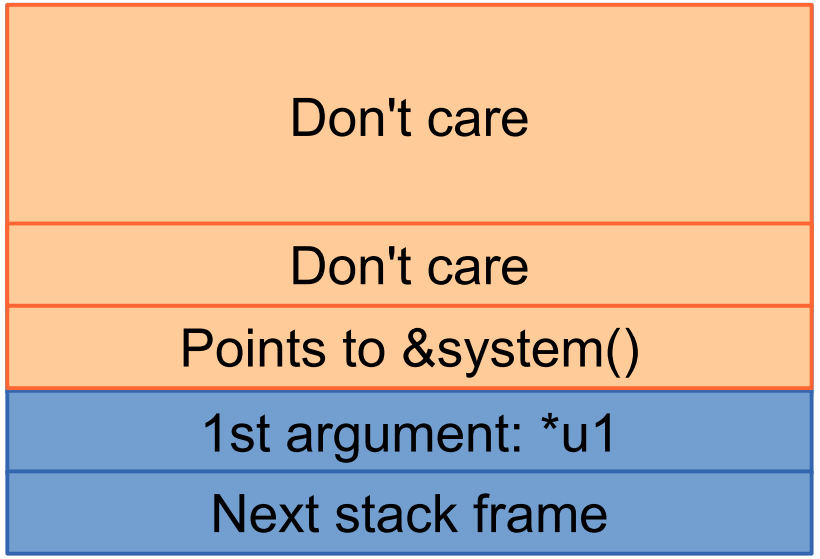
Control-flow hijack attack

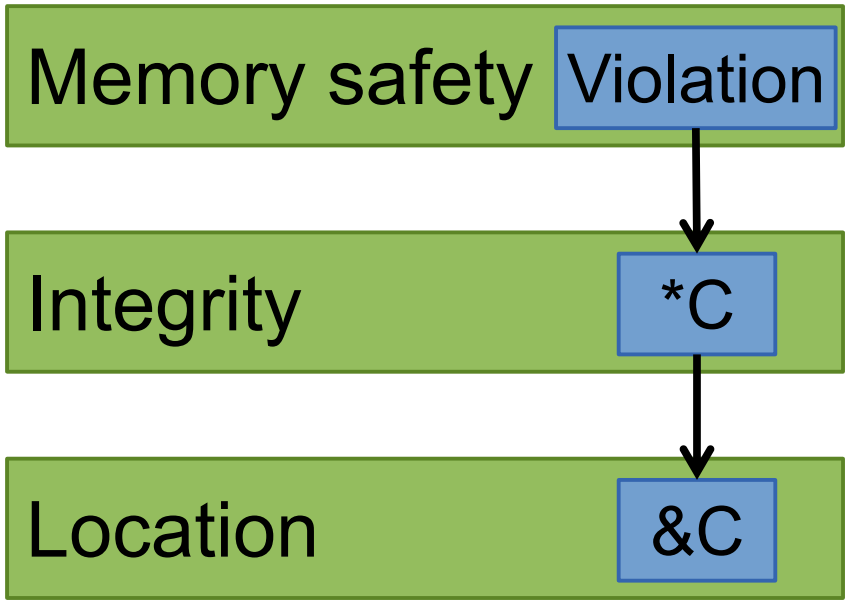
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```



Control-flow hijack attack

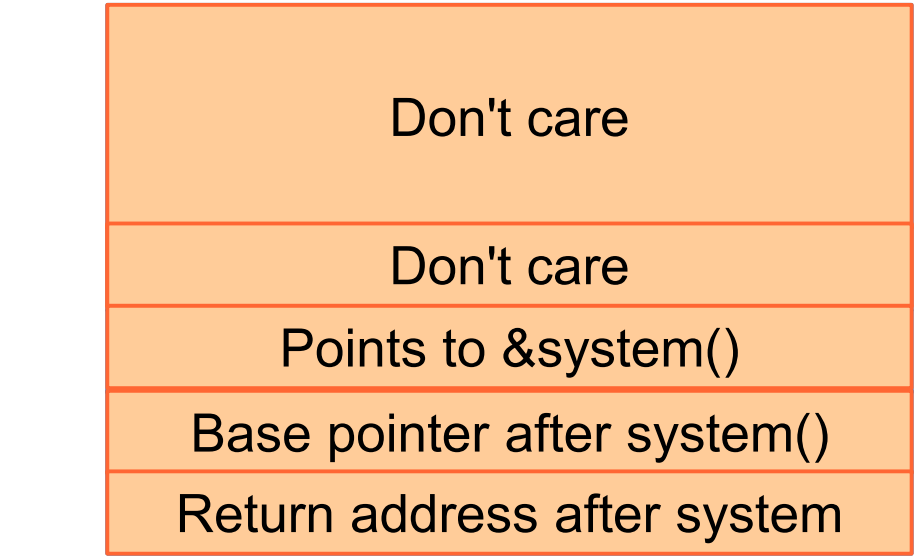
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```

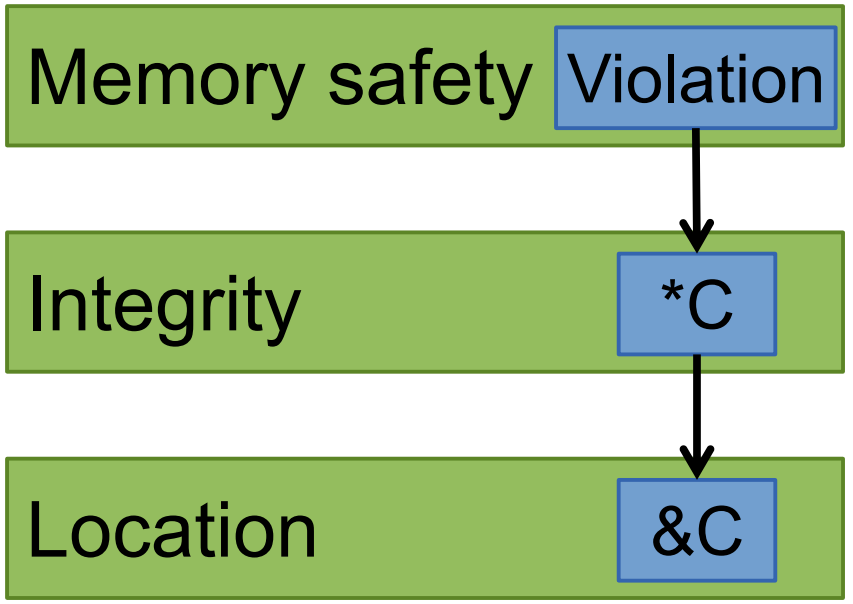




Control-flow hijack attack

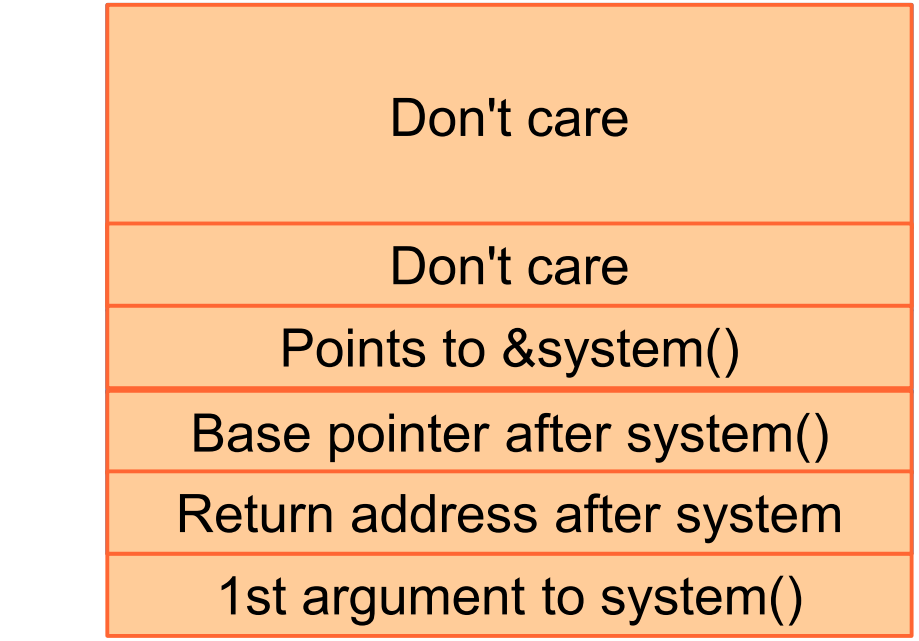
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```

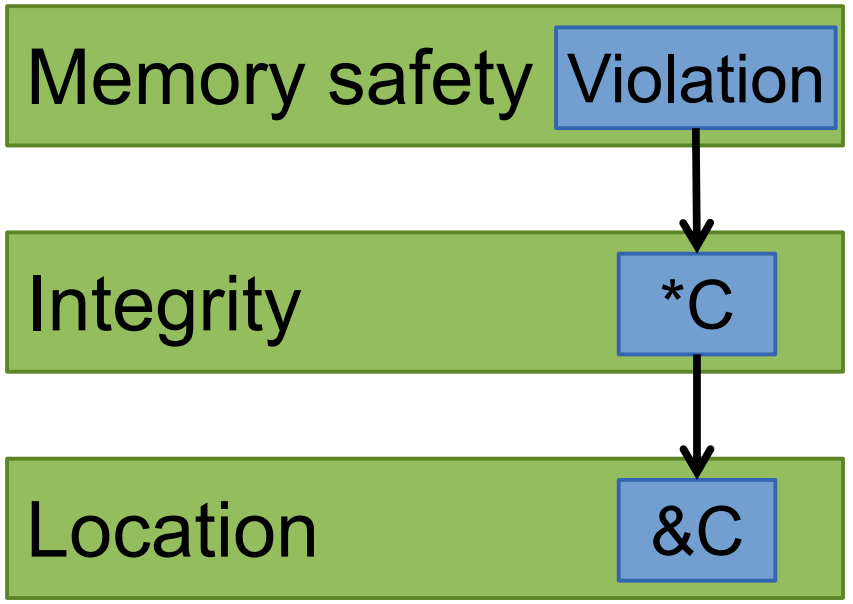




Control-flow hijack attack

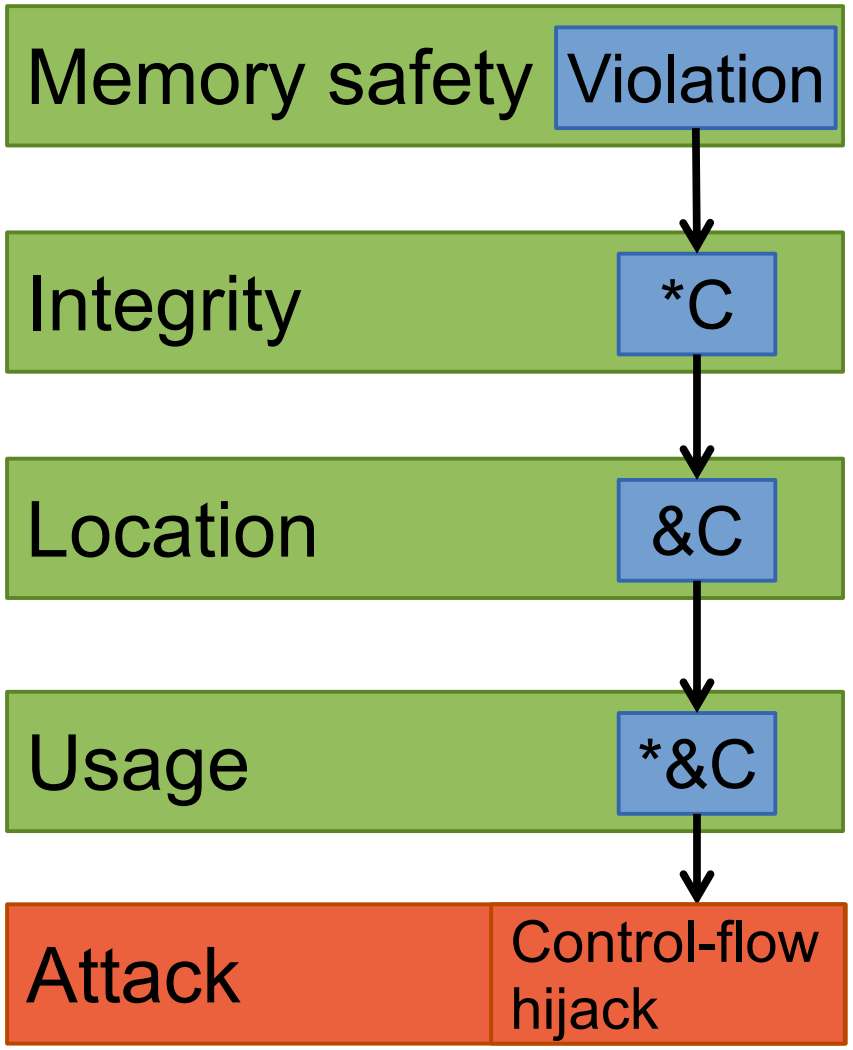
```
void vuln(char *u1) {
        // strlen(u1) < MAX?
        char tmp[MAX];
        strcpy(tmp, u1);
        ...
}
vuln(&exploit);
```





Control-flow hijack attack

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



Points to &system()
Base pointer after system()
Return address after system
1st argument to system()

First Defense: Address Space Layout Randomization

- **Goal**: prevent the attack from reaching a target address

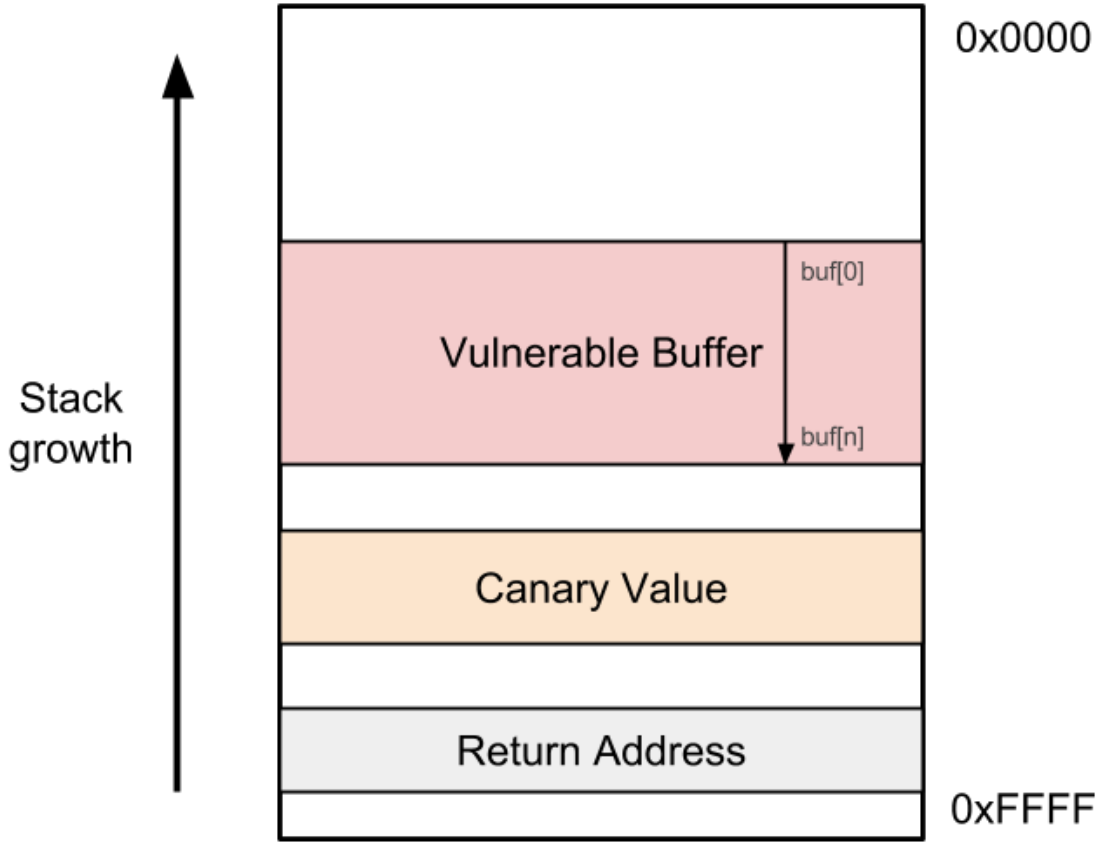
- Randomizes locations of code and data regions
  - Probabilistic defense
  - Depends on loader and OS

- 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)

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

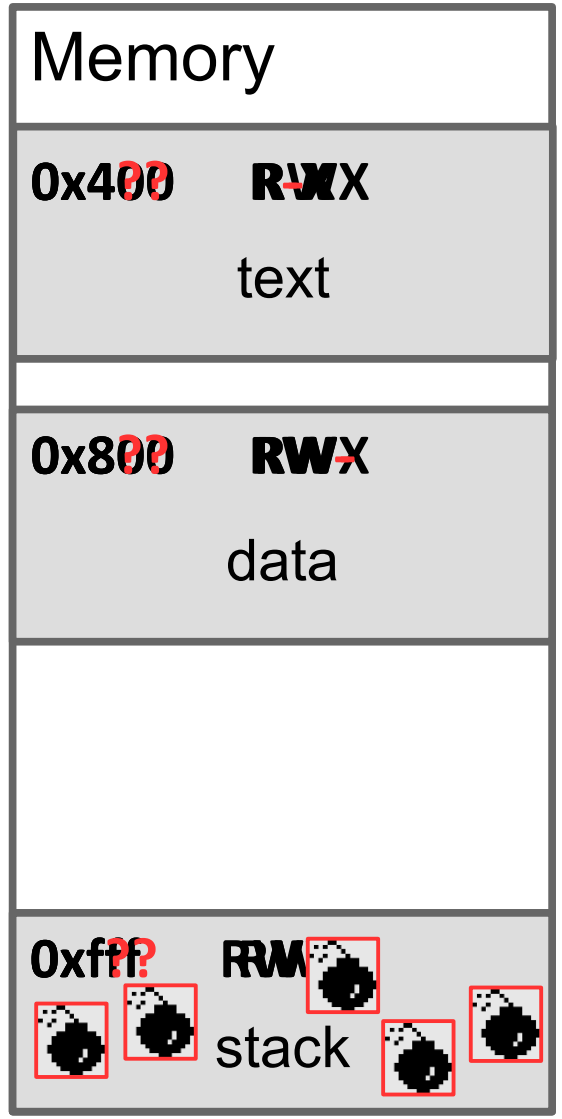
Stack canaries

- 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



Status of deployed defenses

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



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?

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

Control-Flow vs. Data-Flow

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

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

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

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

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

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

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.

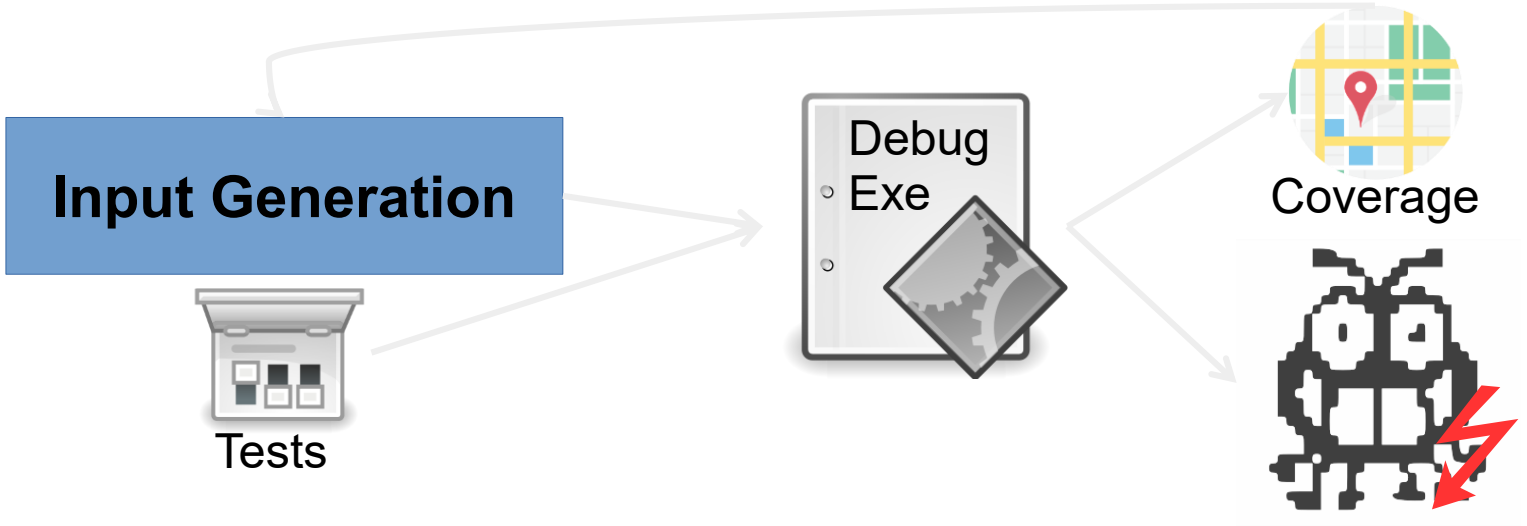
Statement coverage does not imply ***full*** coverage.

Current practice is ***branch coverage***

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.*

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.

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.

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.

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