C programming cheatsheet (preliminaries to follow the lecture)

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

Low-level general-purpose programming language

very efficient

very prevalent (Windows, iOS, IoT)

```
1. #include <stdio.h>
2. int print_hello()
3. {
4. printf("Hello, World!\n");
5. return 0;
6. }
7. x = print_hello()
```

- The function returns an int (→ `int`, line 2)
- Libraries included (other c functions that do not show in the program) (→ line 1)
- Function header (→ line 2)
- Start function (→ line 3)
- Instruction within function (prints Hello World in the screen) (→ line 4)
- Return value "0" (→ line 5)
- End function (→ line 6)
- Store the value returned by print_hello() (→ `x`, line 7)
- Call function (→ line 7)

C language 101: concepts for the lecture

(not a programming course)

```c
int addNumbers(int a, int b)
{
int result;
result = a+b;
return result; // return statement
}
```

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

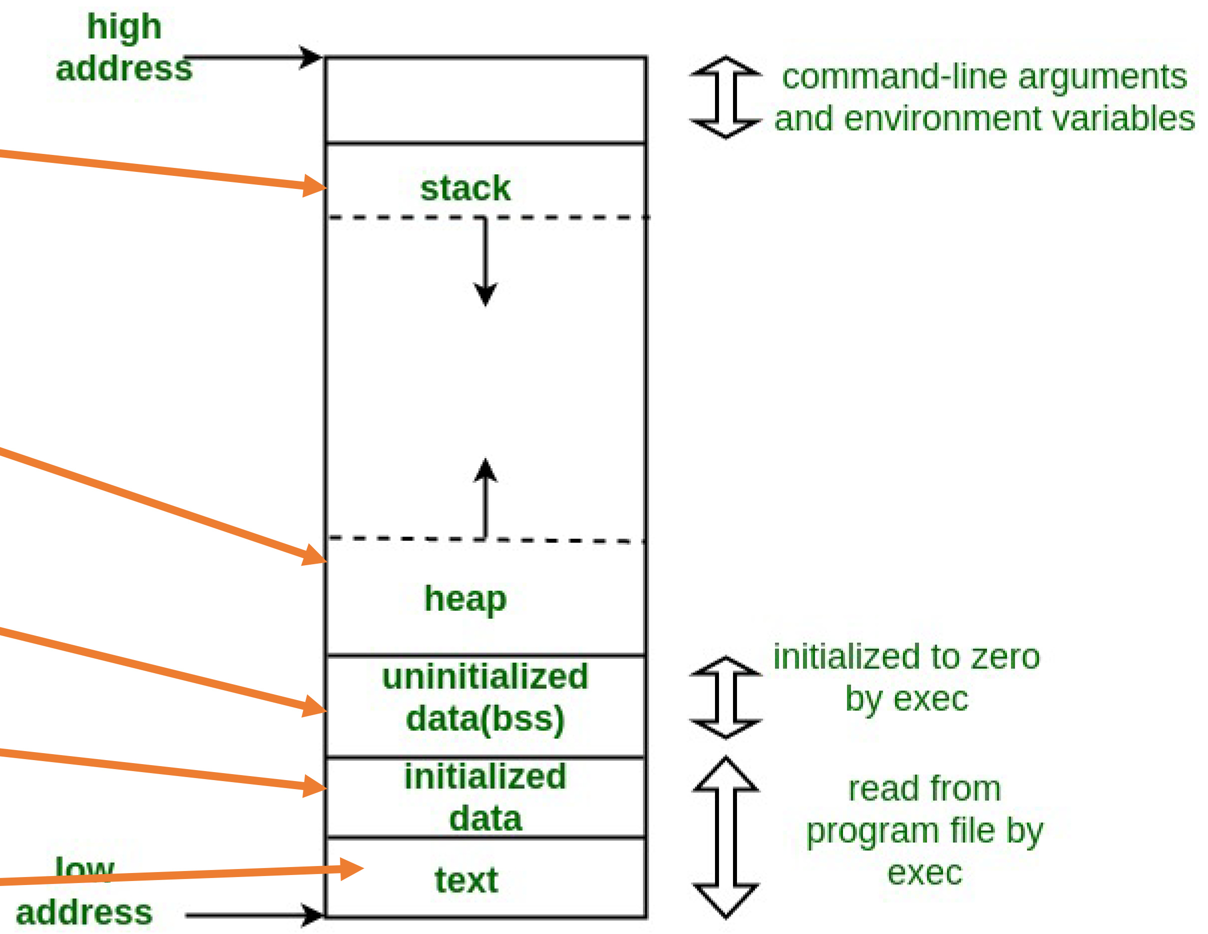
LIFO structure
function variables, along with information that is saved each time a function is called

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

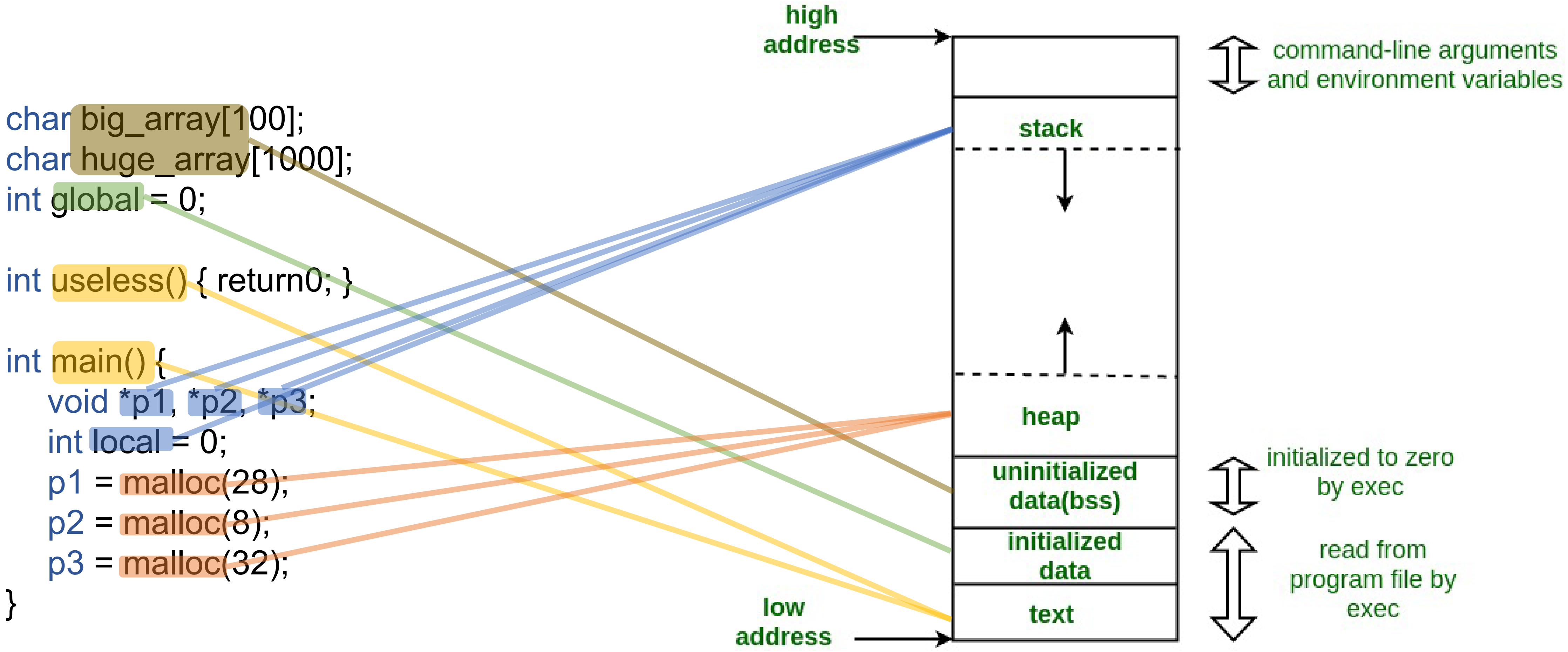
Executable instructions (code). It is under the heap and the stack with the goal of avoiding that it gets rewritten



C language 101: concepts for the lecture

(not a programming course)

Layout of a C program

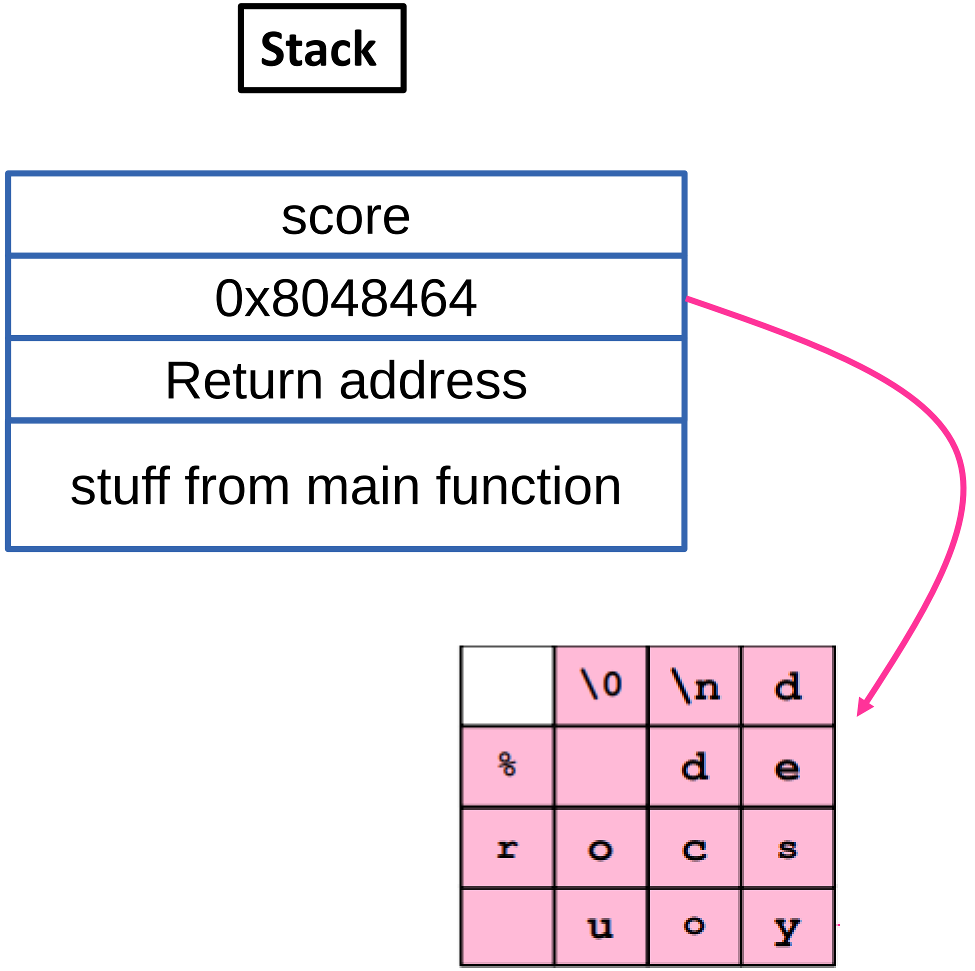


C language 101: concepts for the lecture
(not a programming course)

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 */
}
```



End C programming cheatsheet (preliminaries to follow the lecture)

EPFL

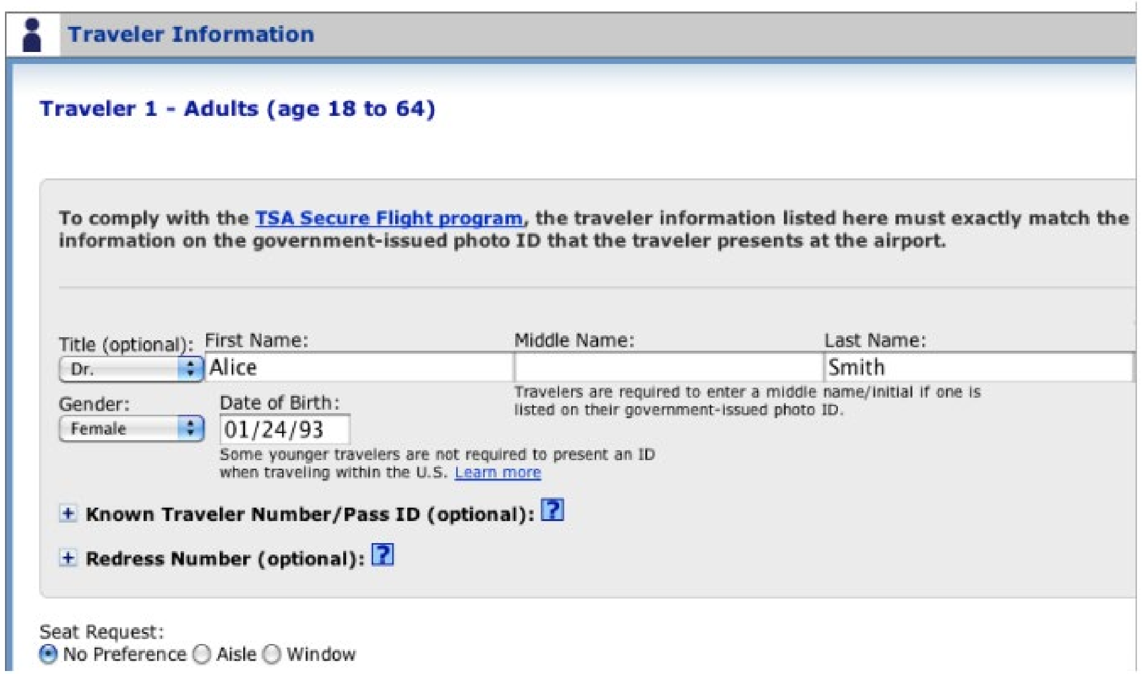
SPRING
SECURITY AND PRIVACY ENGINEERING LABORATORY

Computer Security (COM-301)
Software security
Memory safety

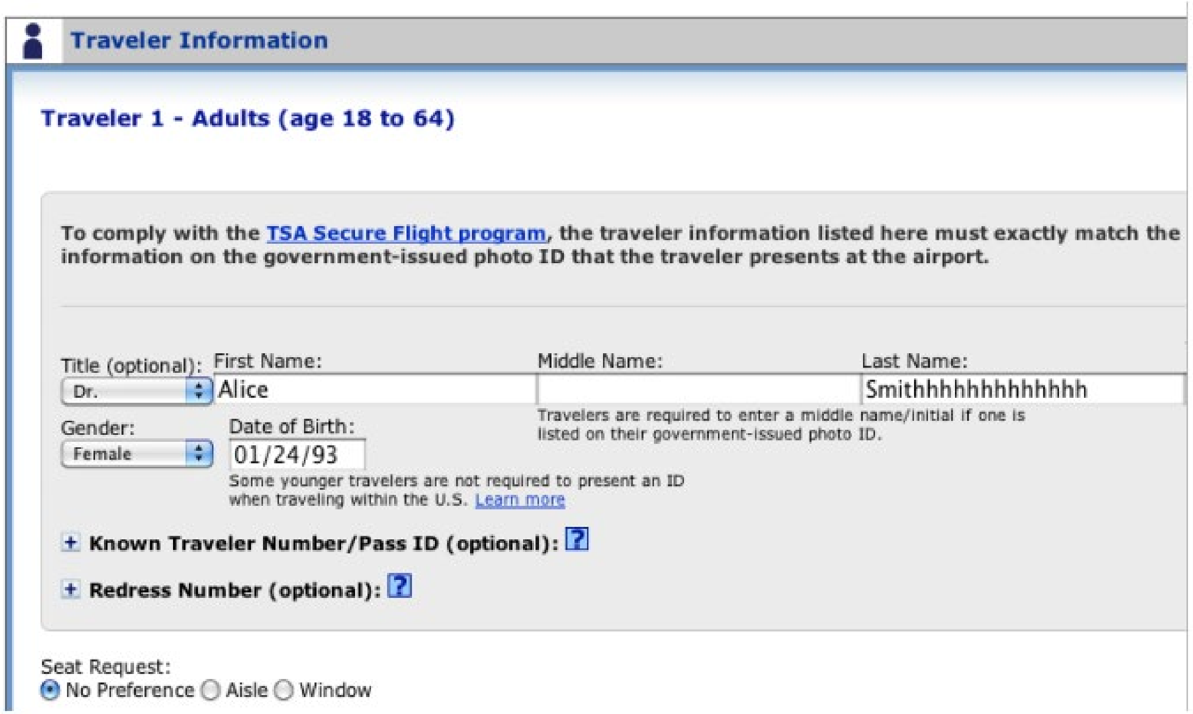
Carmela Troncoso
SPRING Lab
carmela.troncoso@epfl.ch

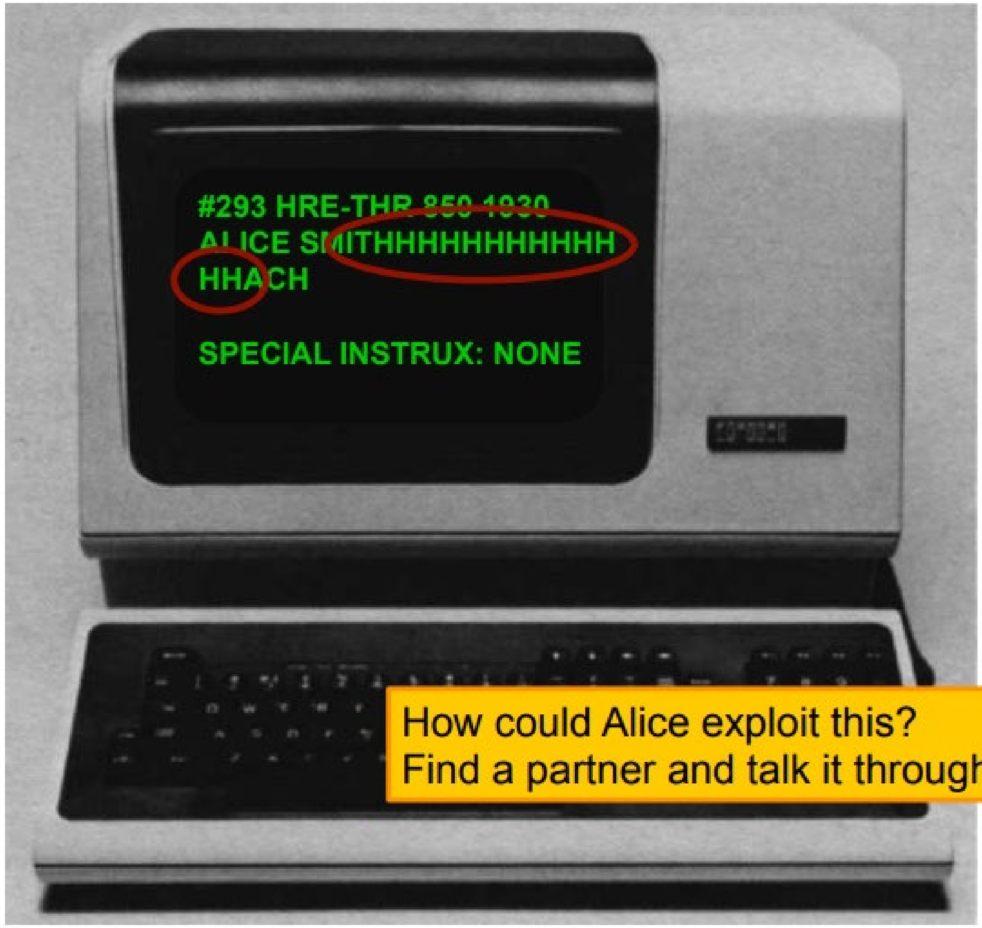
Some slides/ideas adapted from: Tuomas Aura, Yoshi Kohno, Trent Jaeger

Why all the fuzz with overflows…

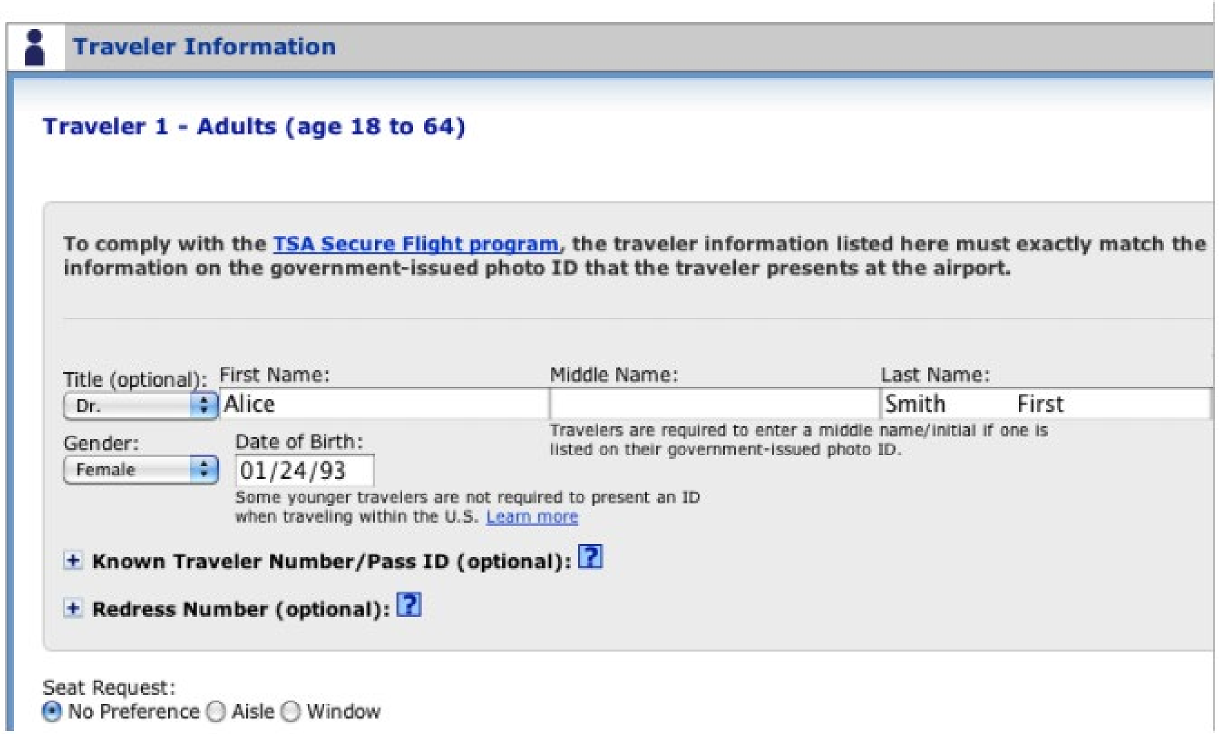

Traveler Information
Traveler 1 - Adults (age 18 to 64)
To comply with the TSA Secure Flight program, the traveler information listed here must exactly match the information on the government-issued photo ID that the traveler presents at the airport.
Title (optional):
Dr.
First Name:
Alice
Middle Name:
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


#293 HRE-THR 850 1930
ALICE SMITH
COACH
SPECIAL INSTRUX: NONE





How could Alice exploit this?
Find a partner and talk it through.


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


#293 HRE-THR 850 1930
ALICE SMITH
FIRST
SPECIAL INSTRUX: NONE

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

Memory corruption

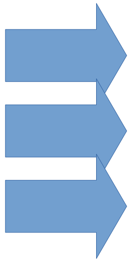
Unintended modification of memory location due to missing / faulty safety check

```
void vulnerable(int user1, int *array) {
        // missing bound check for user1
        array[user1] = 42;
}
```

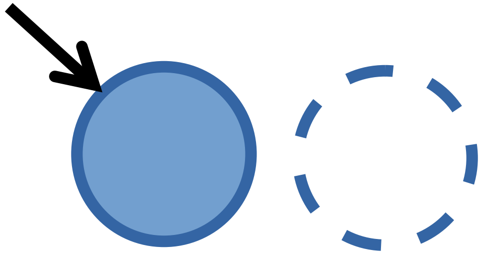
Memory safety: temporal error

```
void vulnerable(char *buf) {
        free(buf);
        buf[12] = 42;
}
```

Memory safety: spatial error



```
void vulnerable() {
        char buf[12];
        char *ptr = buf[11];
        *ptr++ = 10;
        *ptr = 42;
}
```



Memory safety: spatial error

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!

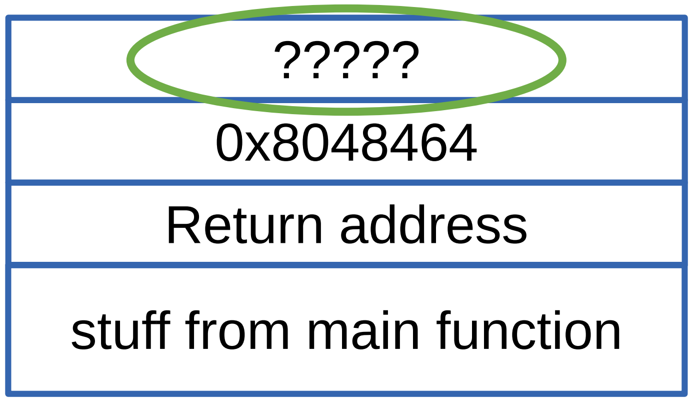
`Gets(buf)`: reads a line from stdin and stores it into the string pointed to by `buf`

Uncontrolled Format String *(CWE-134)*

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

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

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

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

Solving the problem

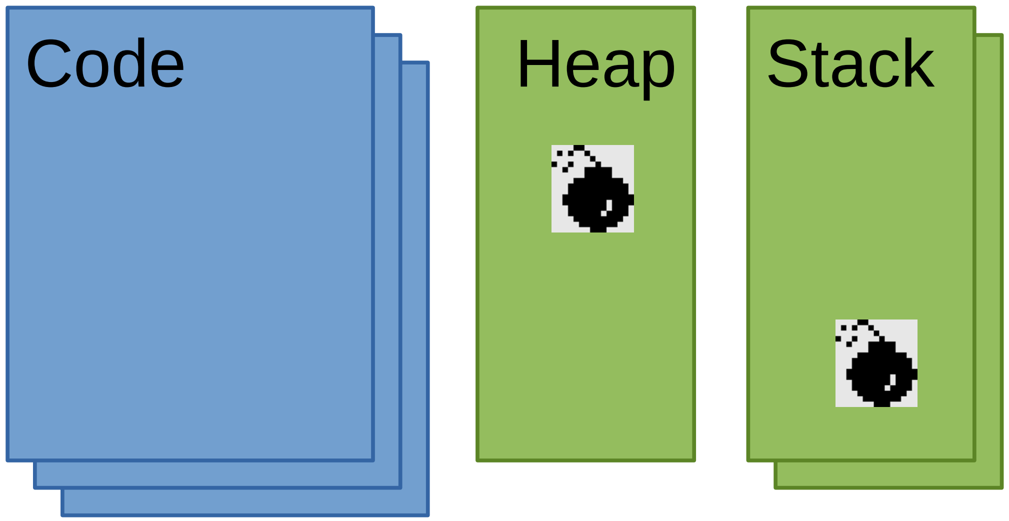
The programmer should decide the format of the string. That ensures that no extra argument, read or write, can be used.

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



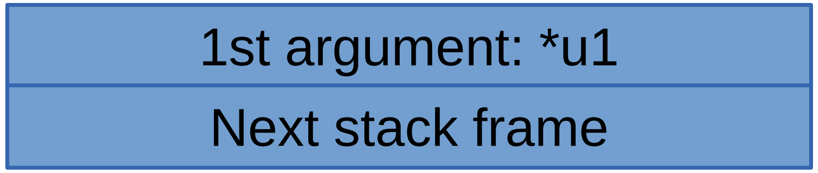
Code injection attack

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

Next stack frame

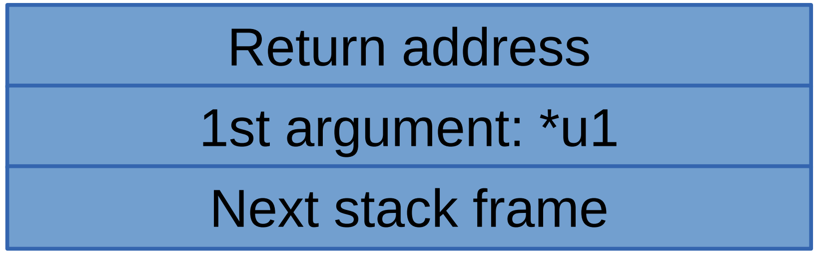
Code injection attack

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



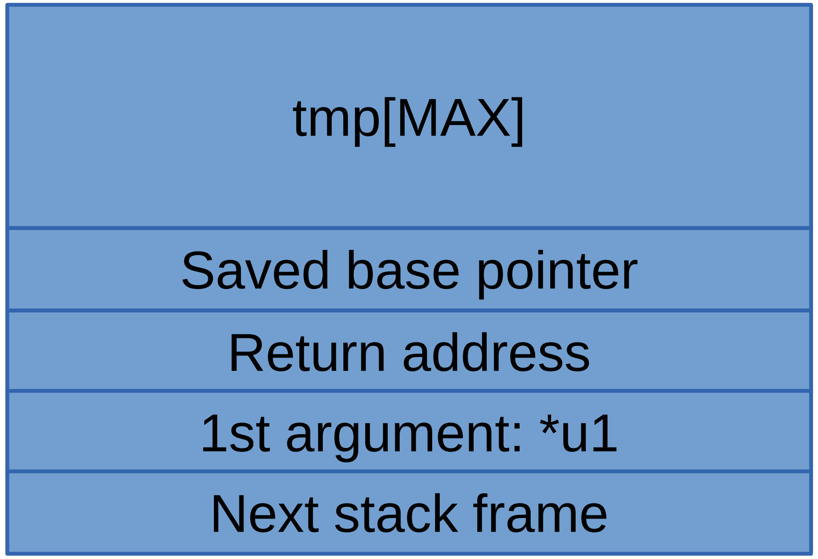
Code injection attack

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



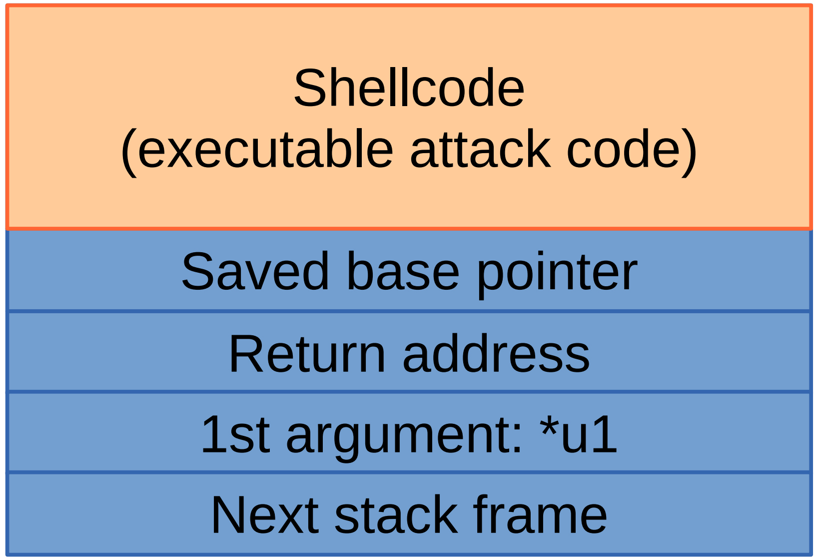
Code injection attack

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



Code injection attack

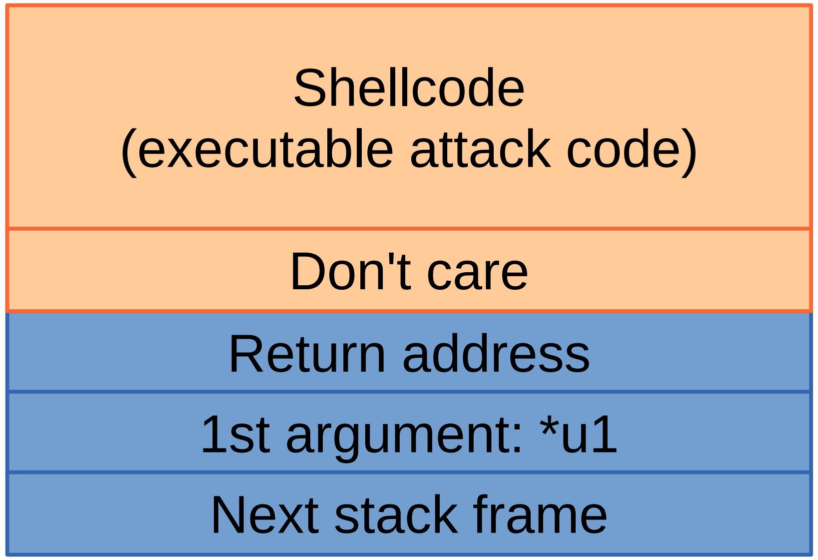
```
void vuln(char *u1) {
      // strlen(u1) < MAX?
      char tmp[MAX];
      strcpy(tmp, u1);
      ...
}
vuln(&exploit);
```



Code injection attack

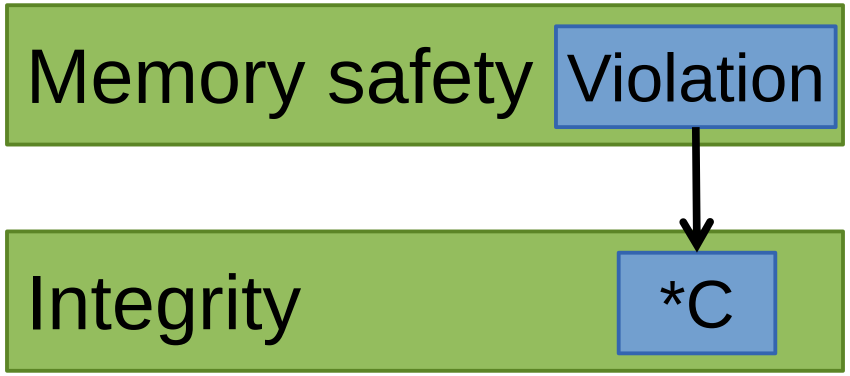
```
void vuln(char *u1) {
      // strlen(u1) < MAX?
      char tmp[MAX];
      strcpy(tmp, u1);
      ...
}
vuln(&exploit);
```

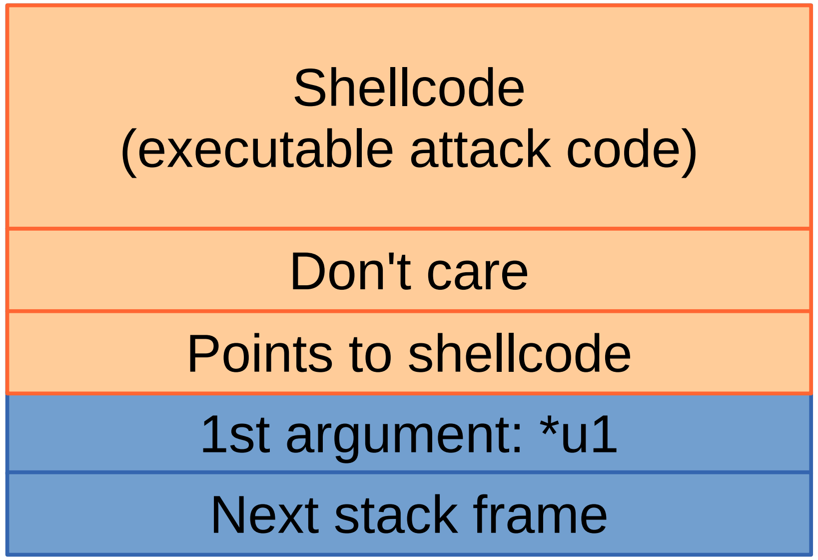
Memory safety Violation



Code injection attack

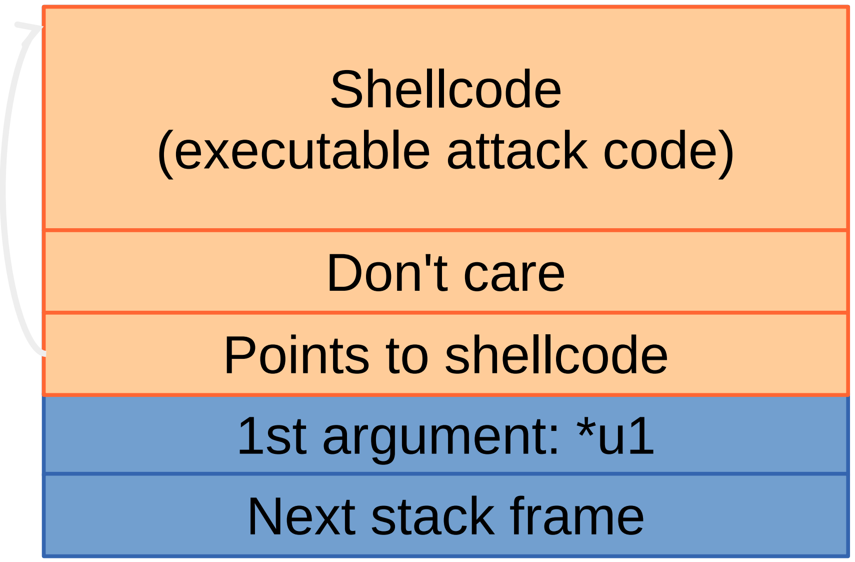
```
void vuln(char *u1) {
      // strlen(u1) < MAX?
      char tmp[MAX];
      strcpy(tmp, u1);
      ...
}
vuln(&exploit);
```

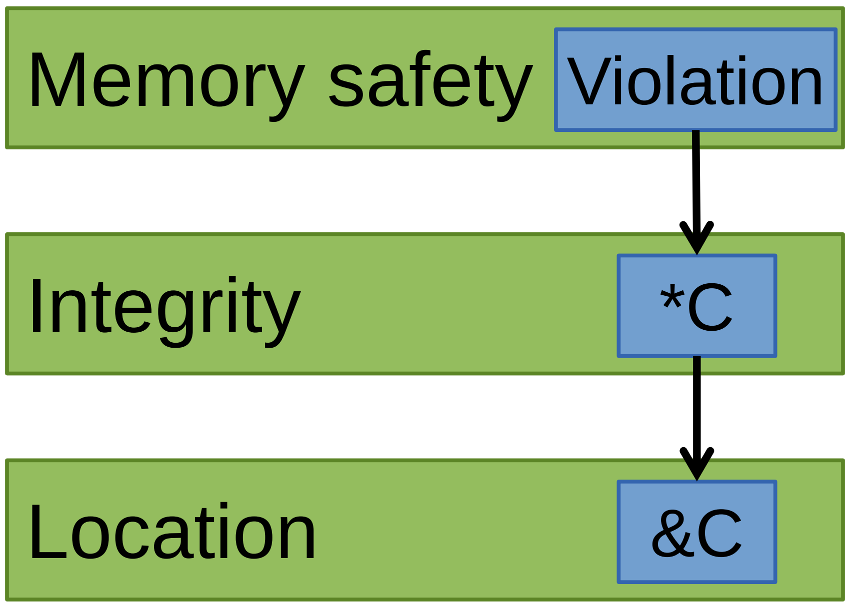




Code injection attack

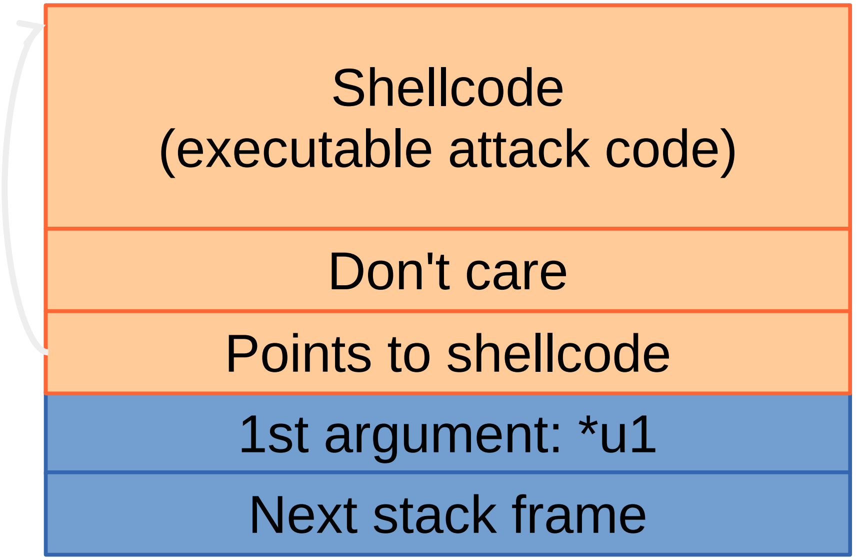
```
void vuln(char *u1) {
      // strlen(u1) < MAX?
      char tmp[MAX];
      strcpy(tmp, u1);
      ...
}
vuln(&exploit);
```

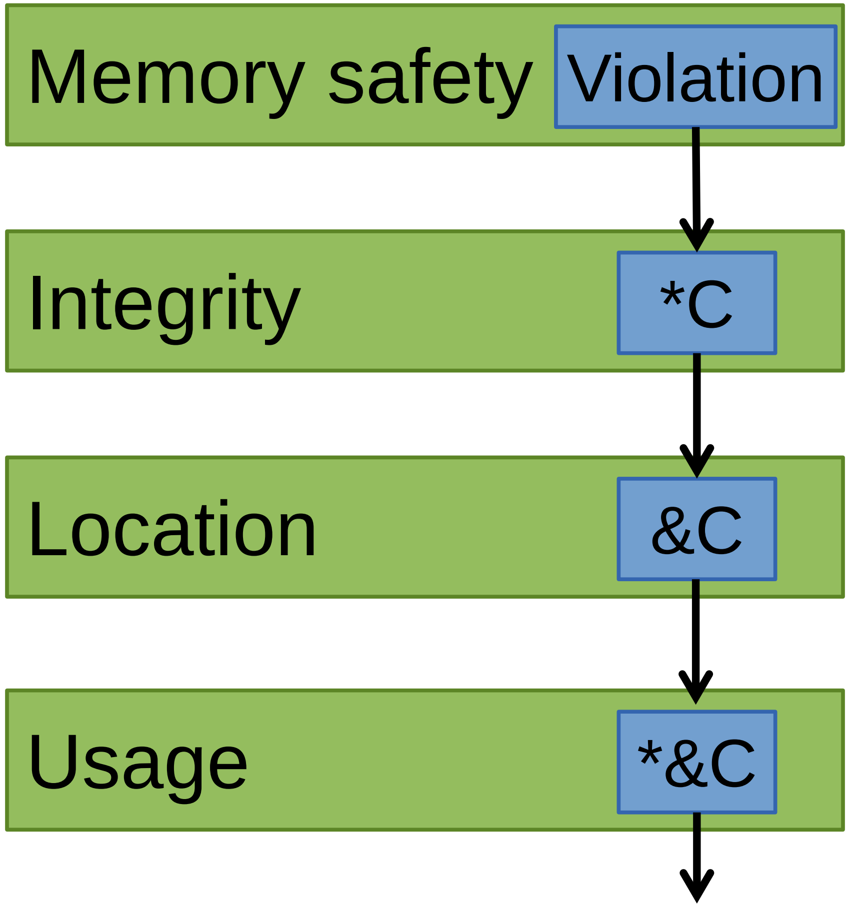




Code injection attack

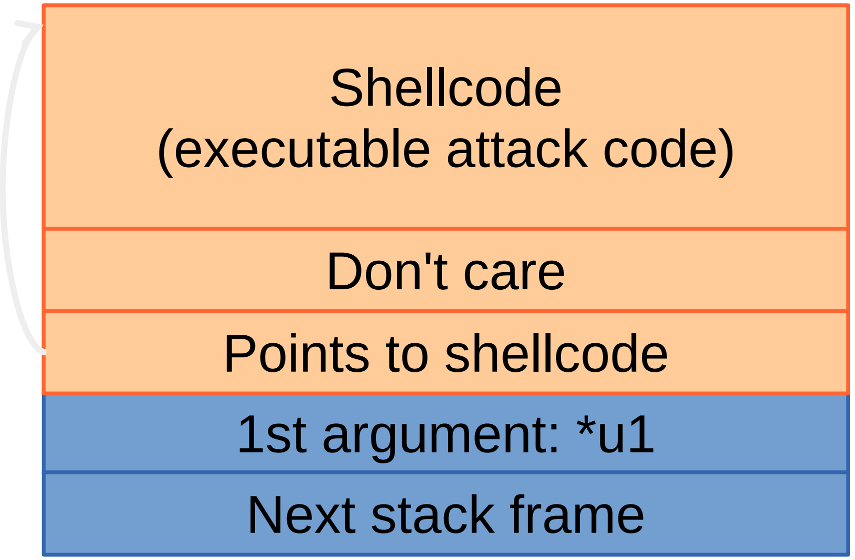
```
void vuln(char *u1) {
      // strlen(u1) < MAX?
      char tmp[MAX];
      strcpy(tmp, u1);
      ...
}
vuln(&exploit);
```

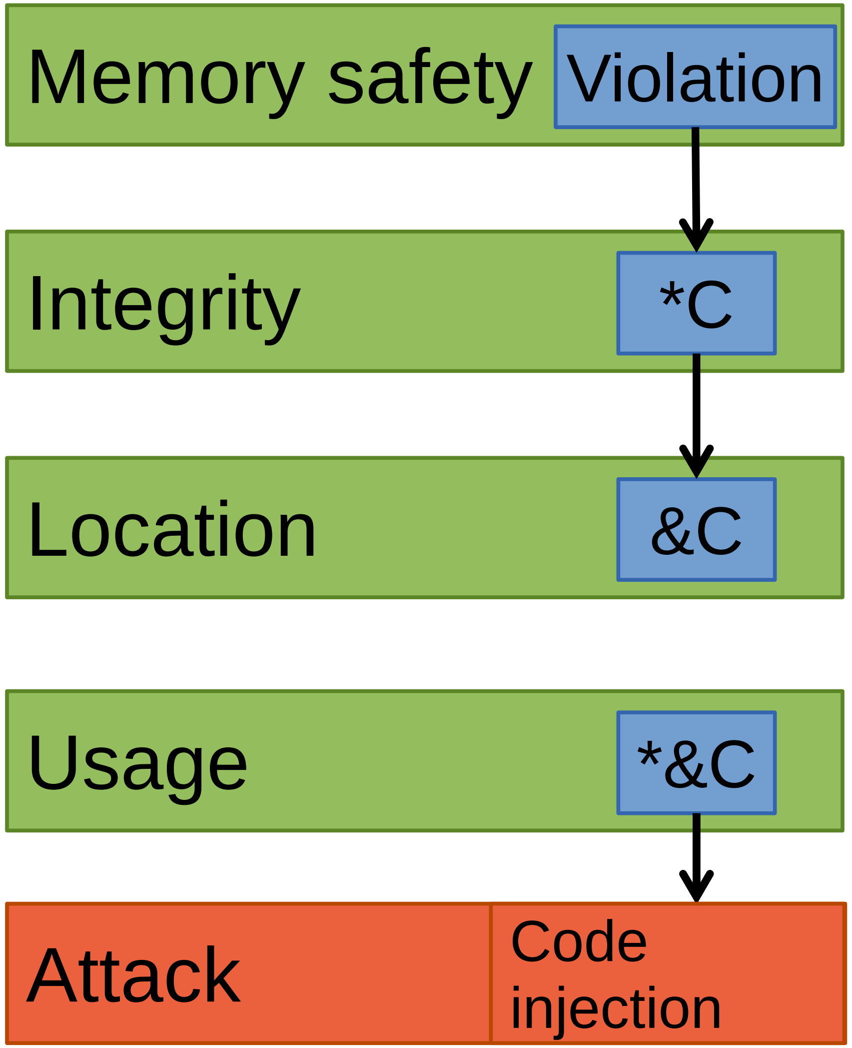




Code injection attack

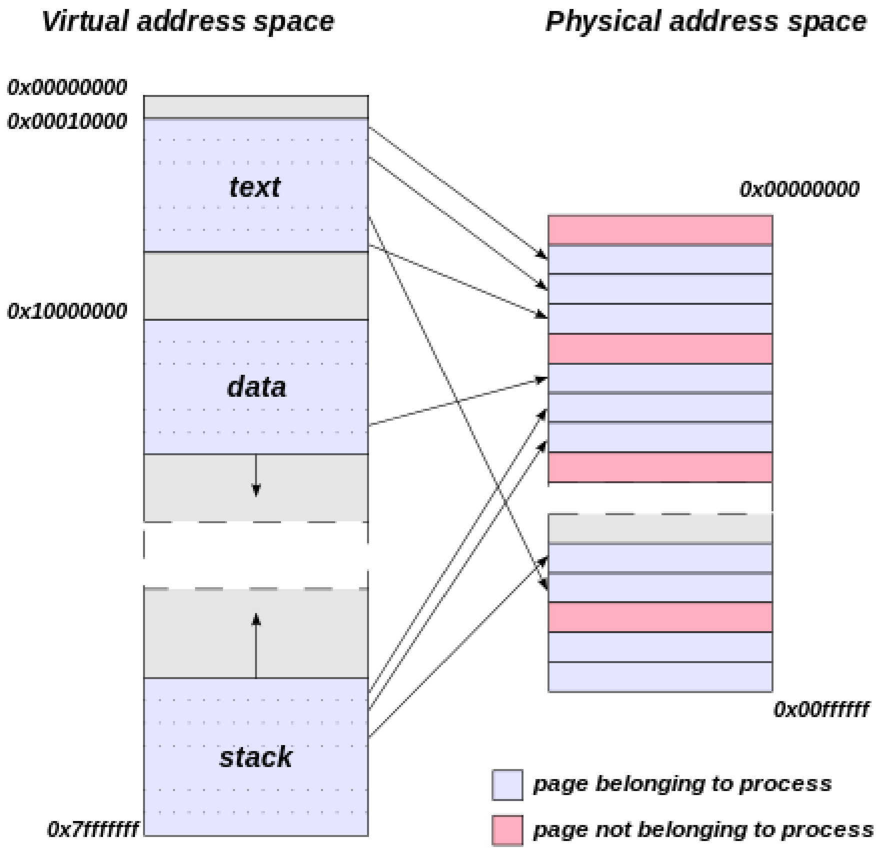
```
void vuln(char *u1) {
      // strlen(u1) < MAX?
      char tmp[MAX];
      strcpy(tmp, u1);
      ...
}
vuln(&exploit);
```





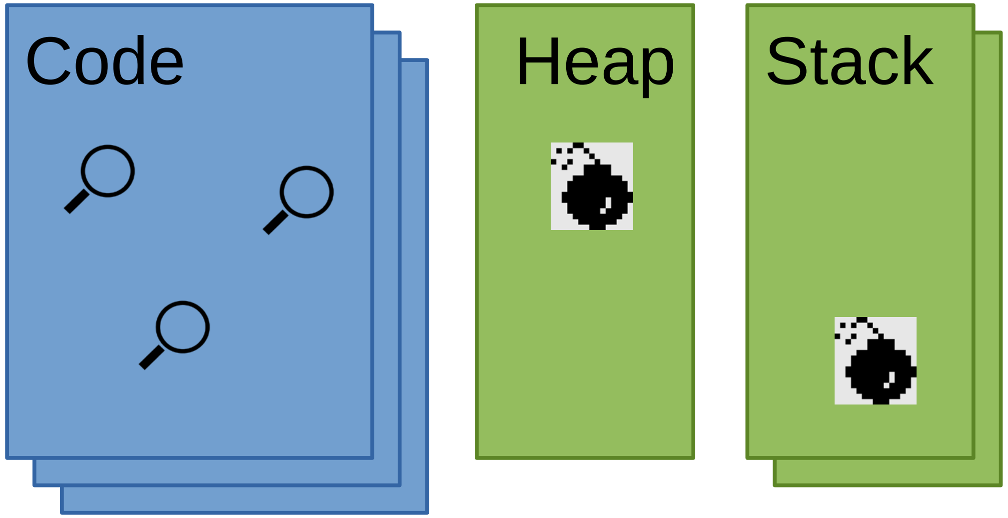
Data Execution Prevention

- 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);
```

1st argument: *u1
Next stack frame

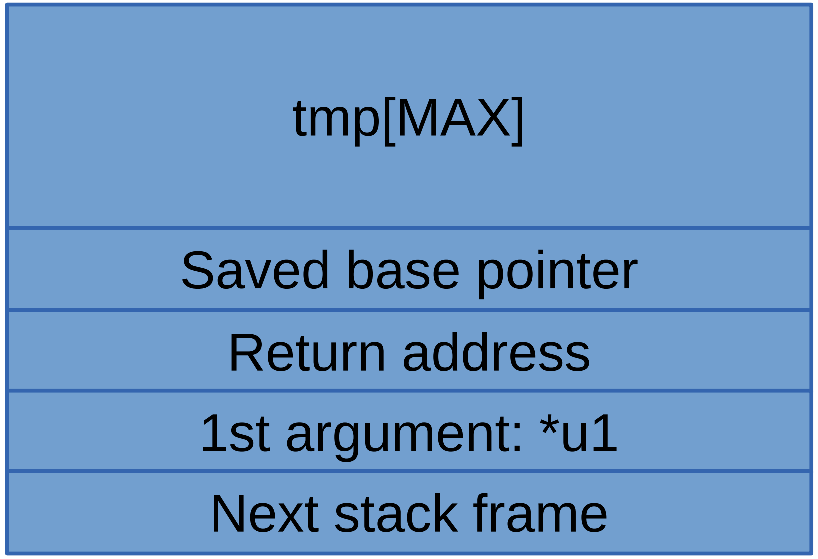
Control-flow hijack attack

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

Saved base pointer
Return address
1st argument: *u1
Next stack frame

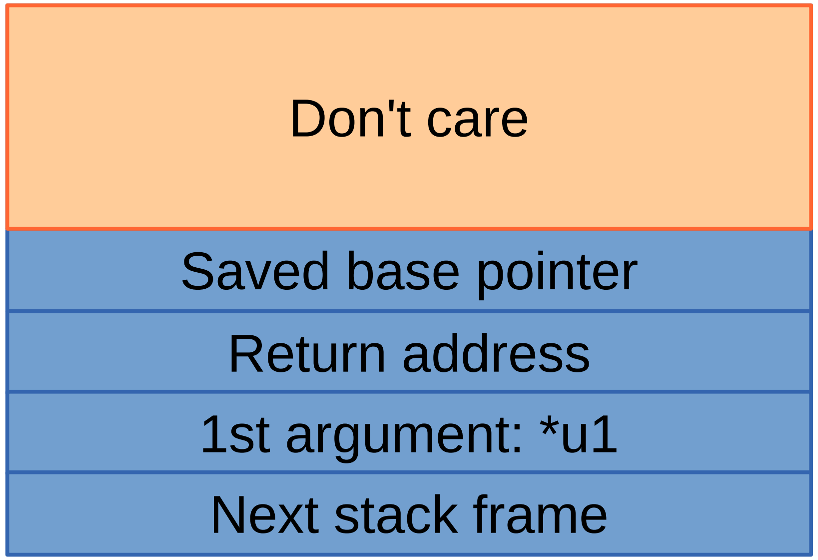
Control-flow hijack attack

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



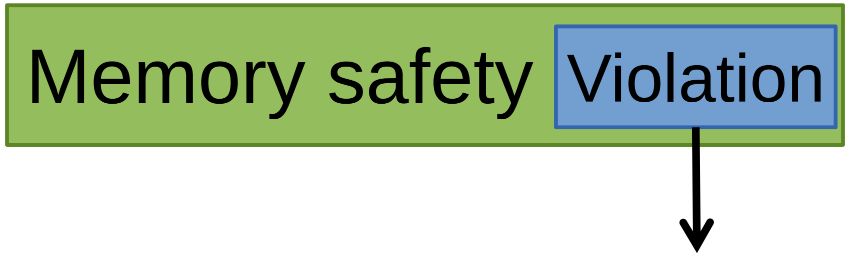
Control-flow hijack attack

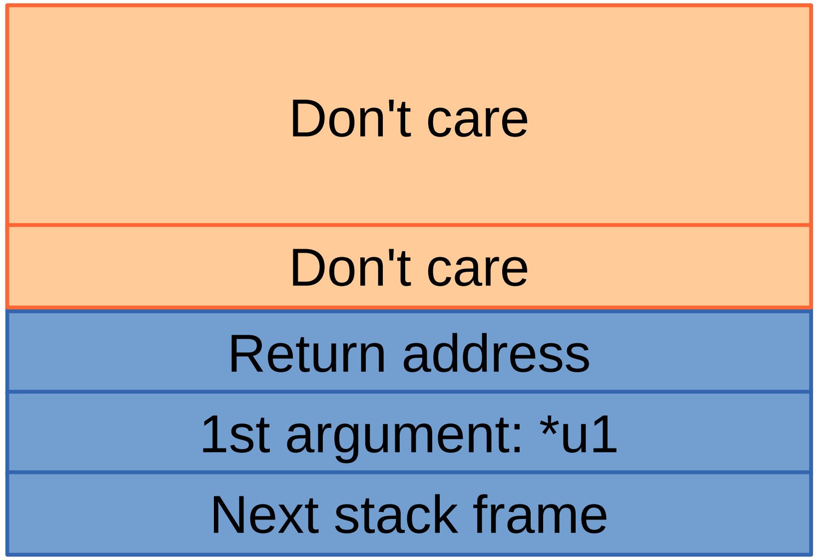
```
void vuln(char *u1) {
      // strlen(u1) < MAX?
      char tmp[MAX];
      strcpy(tmp, u1);
      ...
}
vuln(&exploit);
```



Control-flow hijack attack

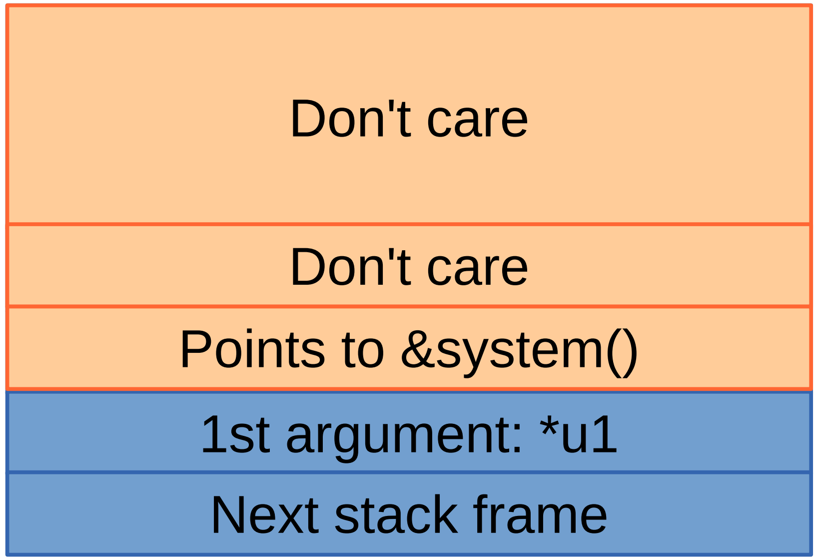
```
void vuln(char *u1) {
      // strlen(u1) < MAX?
      char tmp[MAX];
      strcpy(tmp, u1);
      ...
}
vuln(&exploit);
```

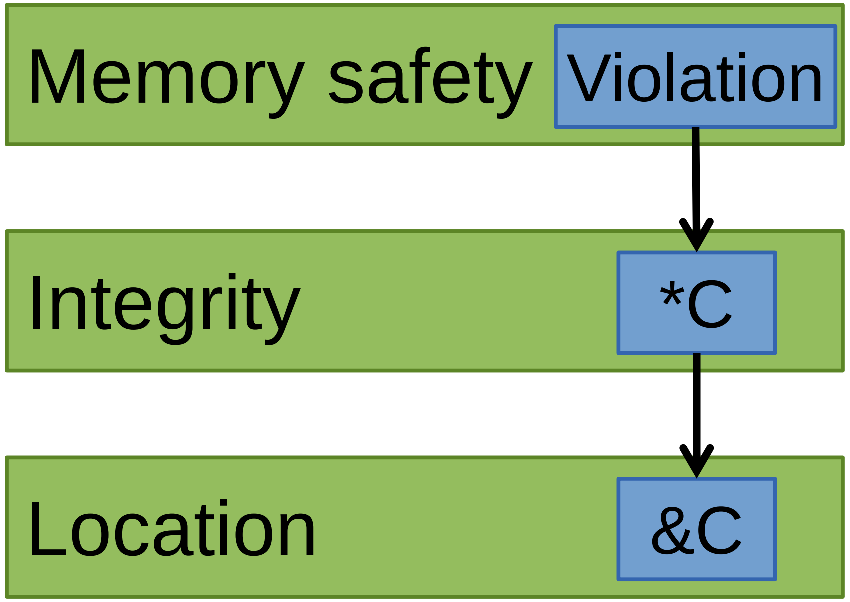




Control-flow hijack attack

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

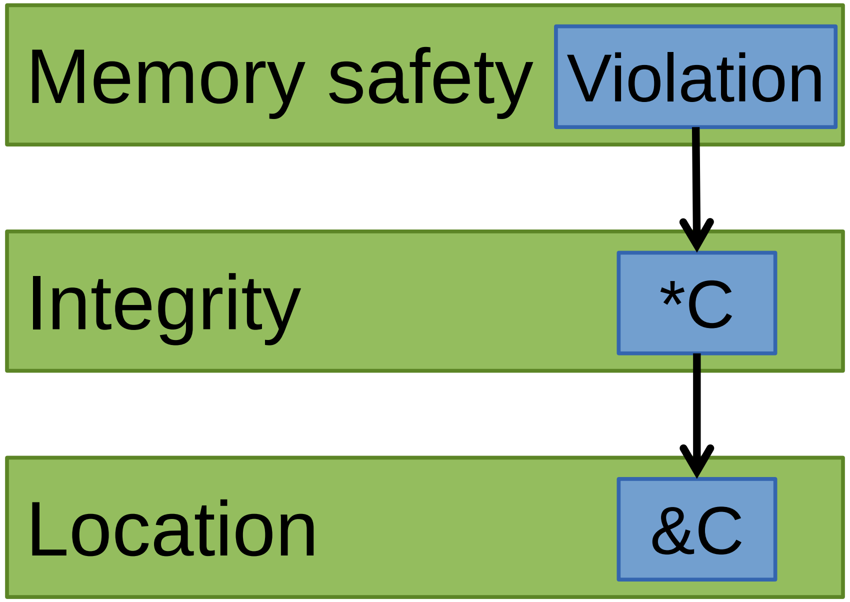




Control-flow hijack attack

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

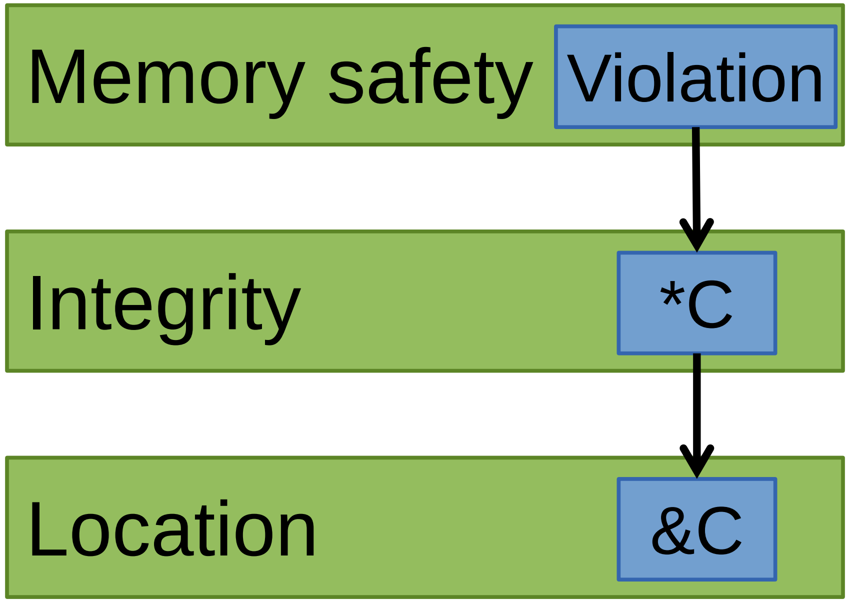
Don't care
Don't care
Points to &system()
Base pointer after system()
Return address after system



Control-flow hijack attack

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

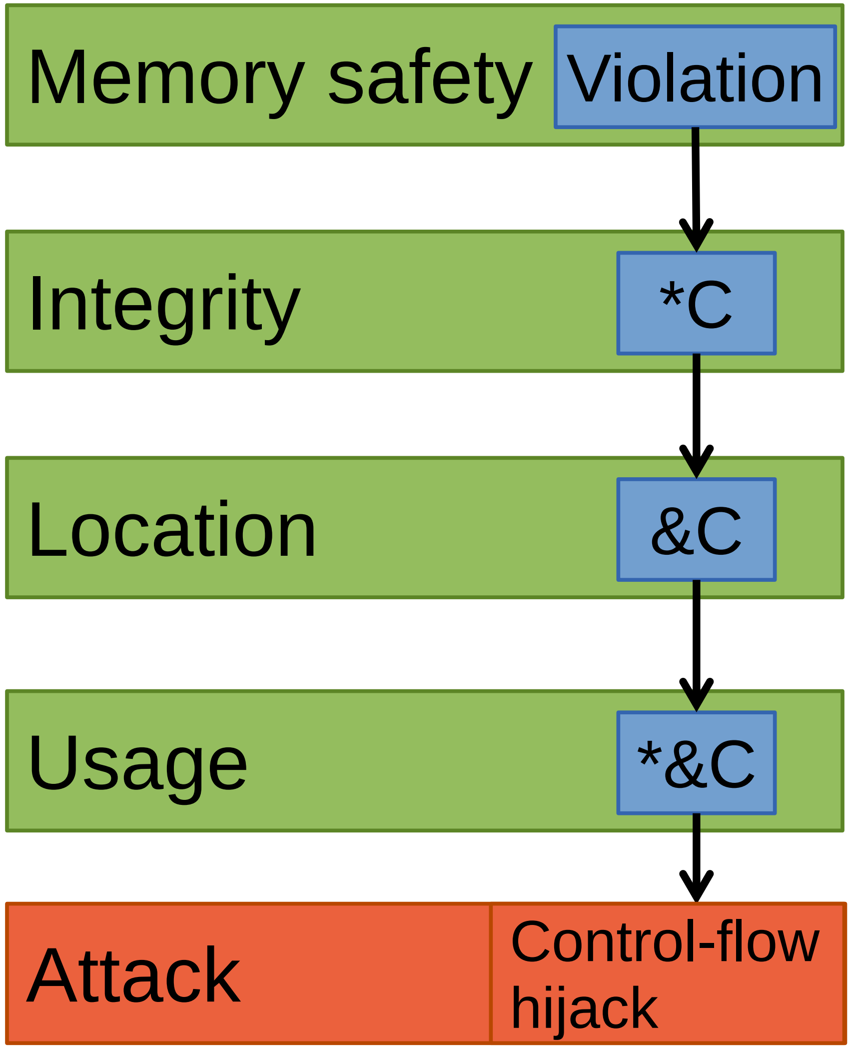
Don't care
Don't care
Points to &system()
Base pointer after system()
Return address after system
1st argument to system()



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()



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

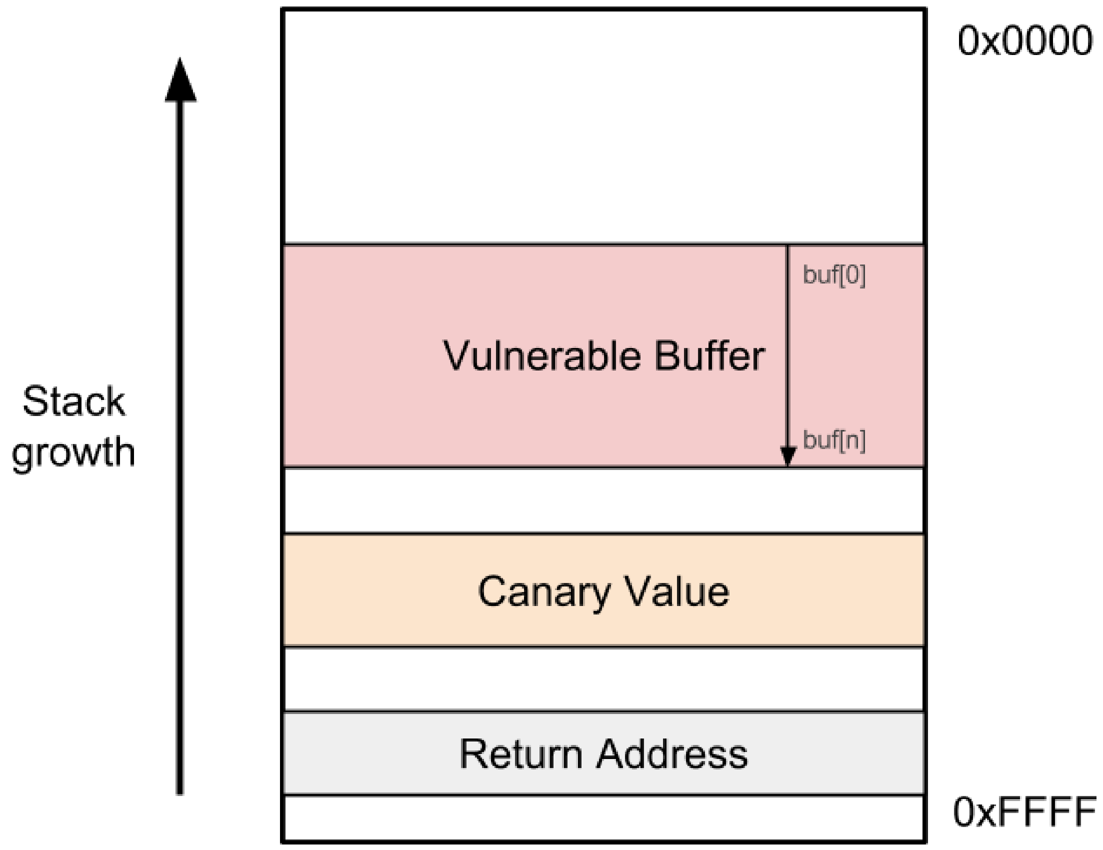
- Weaknesses and limitations
  - Undefined behavior: prone to information leaks
  - Some regions remain static (on x86)
  - Performance impact (~10%)

Stack canaries

Stack canaries

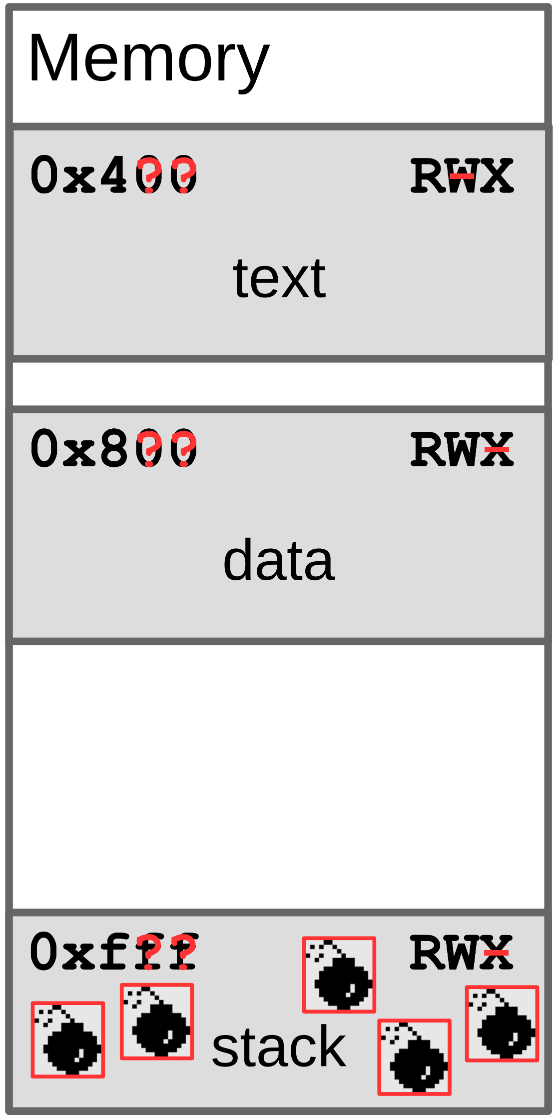
- Protect return instruction pointer on stack
  - Compiler modifies stack layout
  - Probabilistic protection

- 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

Achieving this would be equivalent to solving the "halting problem"

- 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

- Code review
- Heuristic test cases

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.

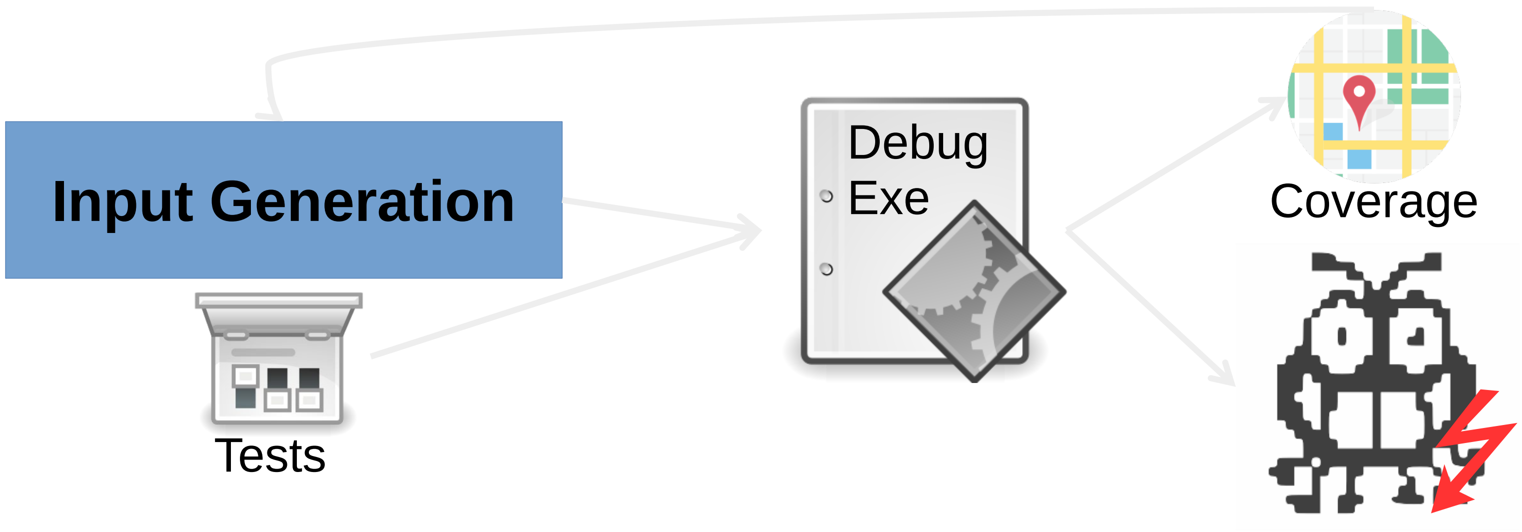
Statement coverage does not imply ***full*** coverage.

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

Two approaches: mitigation and testing

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