IERG 4130 – 2026 Fall

香港中文大學
CUHK

Mitigating Memory Safety Vulnerabilities

Yajin ZHOU

https://yajin.org

Credits: Some of the slides are from CS161 of UC Berkeley, CSE 443 of PSU (Trent Jaeger), CS-412 of EPFL (Mathias Payer), CMPSC 447 of PSU (Gang Tan)

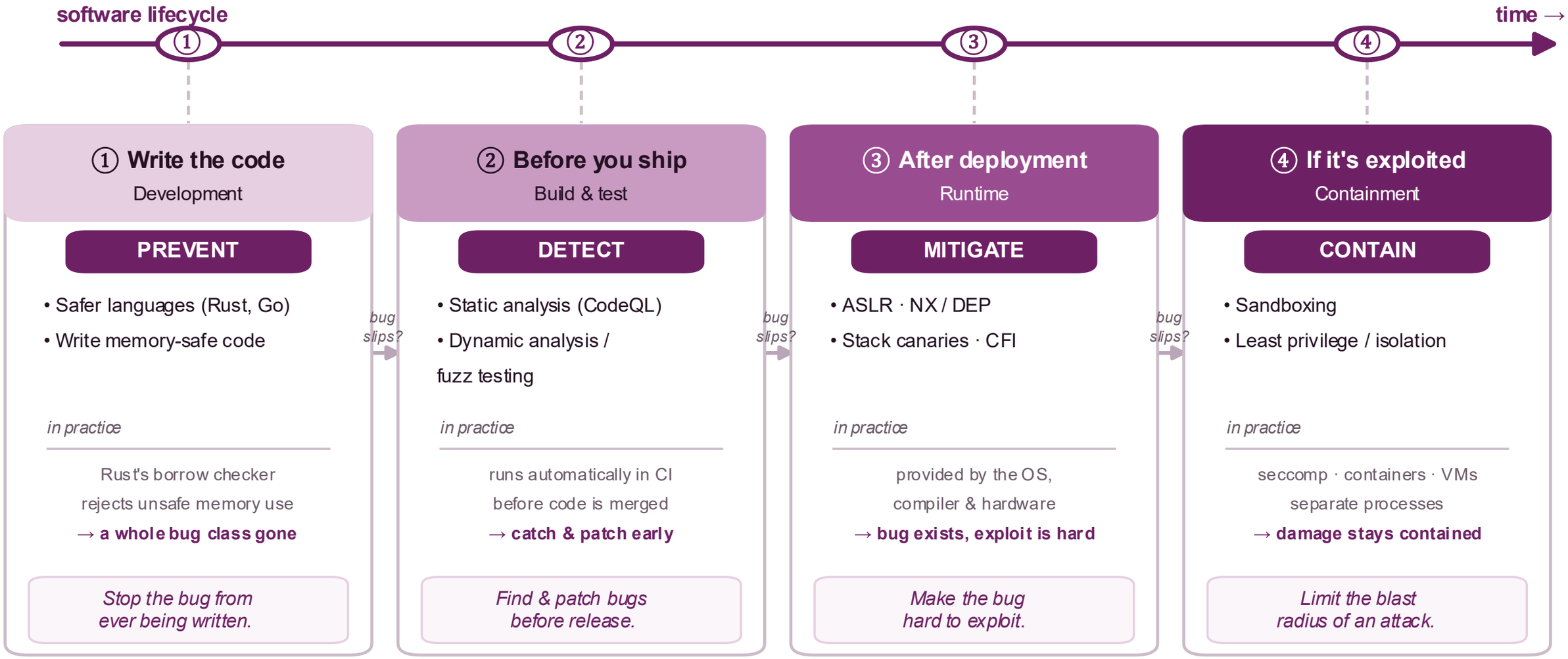
Defending Against Memory Safety Vulnerabilities

- We've seen how widespread and dangerous memory safety vulnerabilities can be. Why do these vulnerabilities exist?
  - Programming languages aren't designed well for security.
  - Programmers often aren't security-aware.
  - Programmers write code without designing security in from the start.
  - Programmers are humans. **Humans make mistakes**.

Different Stages

Defending against memory-safety bugs

different measures act at different stages of the software lifecycle



Defense in depth — each stage is another chance to stop the attack: a bug that slips one layer still meets the next.

香港中文大學
CUHK

Using Memory-Safe Languages

Memory-Safe Languages

- **Memory-safe languages** are designed to automatically **check bounds and prevent undefined memory accesses**
- By design, memory-safe languages are not vulnerable to memory safety vulnerabilities
  - Using a memory-safe language is the **only** way to stop 100% of memory safety vulnerabilities
- Examples: Java, Python, C#, Go, Rust

Memory-Safe Languages

- **Performance:** Extra cost of bounds-checking on every array/memory access
  - Also, extra cost for garbage collection
  - C and C++ (not memory safe): **malloc** usually runs in (amortized) constant-time
  - Java (memory safe): The garbage collector may need to run at any arbitrary point in time, adding a 10–100 ms delay as it cleans up memory

Memory-Safe Languages

- **Legacy**: Huge existing code bases are written in C. Very expensive to rewrite in a new language. Building on existing code is easier than starting from scratch.

- But AI can help!

Memory-Safe Languages

- For many high-level applications, the performance difference from using a memory-safe language is insignificant
  - Possible exceptions: Operating systems, high performance games, some embedded systems
- There are some alternatives for low-level programming with memory safety
  - e.g., Rust

Memory-Safe Languages

- Writing in higher-level (memory-safe) languages may save developer time
- Sometimes there is a small performance-critical core that can be **encapsulated**
  - e.g., a fast library implemented in C (NumPy) and written carefully to be memory-safe, and then you can invoke the library from a memory-safe language

香港中文大學
CUHK

Writing Memory-Safe Code

Writing Memory-Safe Code

- Defensive programming: Always add checks in your code just in case
  - Example: Always check a pointer is not null before dereferencing it, even if you're sure the pointer is going to be valid
  - Before writing to an array or buffer, check that the write operation will be in-bounds
  - Relies on programmer discipline and tracking the length of every array/buffer/memory region

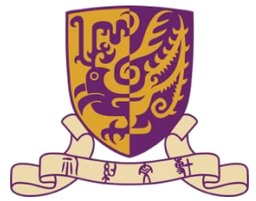
Writing Memory-Safe Code

- Use safe libraries (that do these checks for you)
  - Use functions that check bounds
  - Example: Use **fgets** instead of **gets**
  - Example: Use **strncpy** or **strlcpy** instead of **strcpy**
  - Example: Use **snprintf** instead of **sprintf**
  - Relies on programmer discipline or tools that check your program

Writing Memory-Safe Code

- Reason carefully about your code
  - When writing code, define a set of *preconditions*, *postconditions*, and *invariants* that must be satisfied for the code to be memory-safe
  - Very **tedious** and rarely used in practice, so it's out of scope for this class

gets: Unbounded String Copies



- Occurs when data is copied from an unbounded source to a fixed-length character array

```
void main(void) {
    char Password[8];
    puts("Enter a 8-character password:");
    gets(Password);
    printf("Password=%s\n",Password);
}
```

Better String Library Functions

- Functions that restrict the number of bytes are often recommended
- Never use gets(buf)
  - Use fgets(buf, size, stdin) instead
    - **buf:** Pointer to the character array where the input string is stored.
    - **size:** Maximum number of characters to read, including the null terminator (\0).
    - **stdin:** The input source, such as stdin for keyboard input

From gets to fgets

- char *fgets(char *BUF, int N, FILE *FP);
  - *“Reads at most N-1 characters from FP until a newline is found. The characters up to and including the newline are stored in BUF. The buffer is terminated with a 0.”*

```
void main(void) {
    char Password[8]; 9
    puts("Enter a 8-character password:");
    fgets(Password, 8, stdin);
    ...                       9
}
```

Better String Library Functions

- Instead of strcpy(), use strncpy()
- Instead of strcat(), use strncat()
- Instead of sprintf(), use snprintf()

But Still Need Care

- char *strncpy(char *s1, const char *s2, size_t n);
  - Copy not more than n characters (including the null character) from the array pointed to by s2 to the array pointed to by s1;
  - If the string pointed to by s2 is shorter than n characters, null characters are appended to the destination array until a total of n characters have been written.
- What happens if the size of s2 is n or greater
  - It gets truncated
  - **And s1 may not be null-terminated!**

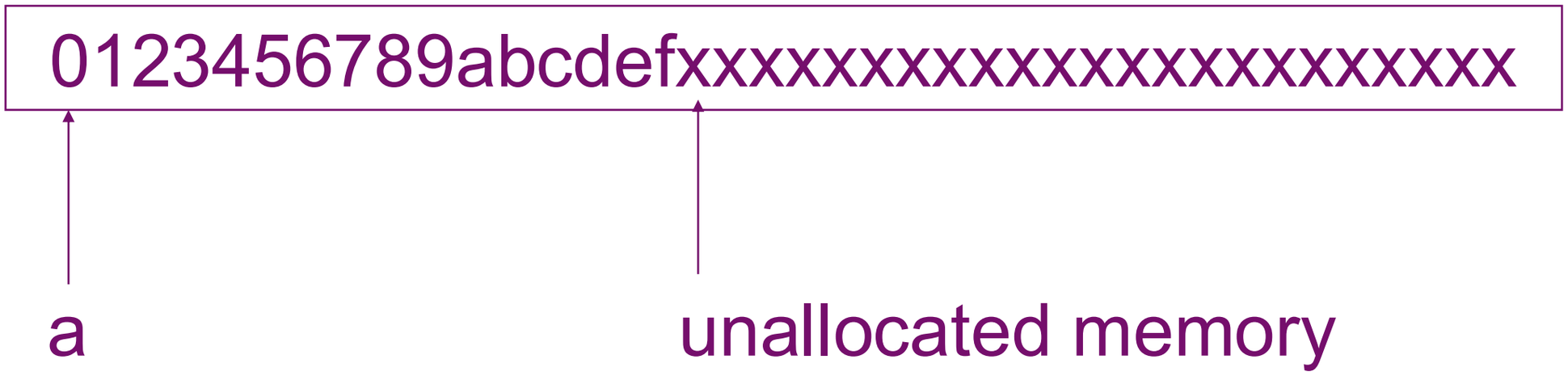
Null-Termination Errors

```
int main(int argc, char* argv[]) {
    char a[16], b[16];
    strncpy(a, "0123456789abcdef", sizeof(a));
    printf("%s\n",a);
    strcpy(b, a);
}
```

- a[] not properly terminated. Possible segmentation fault if printf("%s\n",a);
  - Why?

Null-Termination Errors

- a[] not properly terminated. Possible segmentation fault if printf(“%s\n”,a);
  - Why?



Safe or Vulnerable

```
void func(int len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```

Safe or Vulnerable

```
void func(size_t len, char *data) {
    char *buf = malloc(len + 2);
    if (buf == NULL)
        return;
    memcpy(buf, data, len);
    buf[len] = '\n';
    buf[len + 1] = '\0';
}
```

香港中文大學
CUHK

Building Secure Software

Approaches

- Run-time checks
  - Automatic bounds-checking
  - May involve performance overhead
  - Crash if the check fails

Approaches

- Bug-finding tools
  - Excellent resource, as long as there aren’t too many false alarms
- Code review
  - Have someone look over your code for memory safety errors
  - Can be very effective… but also expensive

Approaches

- Vulnerability scanning
  - Probe your systems for known flaws
- Penetration testing (“pen-testing”)
  - Pay someone to break into your system

Testing for Software Security Issues

- How can we test programs for memory safety vulnerabilities?
  - **Fuzz testing**: Random inputs
  - Use tools like **Valgrind** (tool for detecting memory leaks)
  - Test corner cases
- How do we tell if we've found a problem?
  - Look for a crash or other unexpected behavior
- How do we know that we've tested enough?
  - Hard to know, but **code-coverage tools** can help

Testing for Software Security Issues

Testing is the process of executing a program to find errors.

- An error is a deviation between observed behavior and specified behavior, i.e., a violation of the underlying specification:
  - Functional requirements (features)
  - Operational requirements (performance, usability)
  - **Security requirements**

ref: Chauhan, R.K. and Singh, I., 2014. Latest research and development on software testing techniques and tools.

Testing for Software Security Issues

- Manual testing
  - Unit testing (individual modules)
  - Integration testing (interaction between modules)
  - System testing (full application testing)
- **Fuzz testing**
- Symbolic and concolic testing

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Fuzzing

- At Its Core, Fuzzing is Random Testing
  - It started a long time ago
  - automated software testing technique

1981 Random testing is a cost-effective alternative to systematic testing techniques (Duran & Natos)

1983 "The Monkey" (Capps)

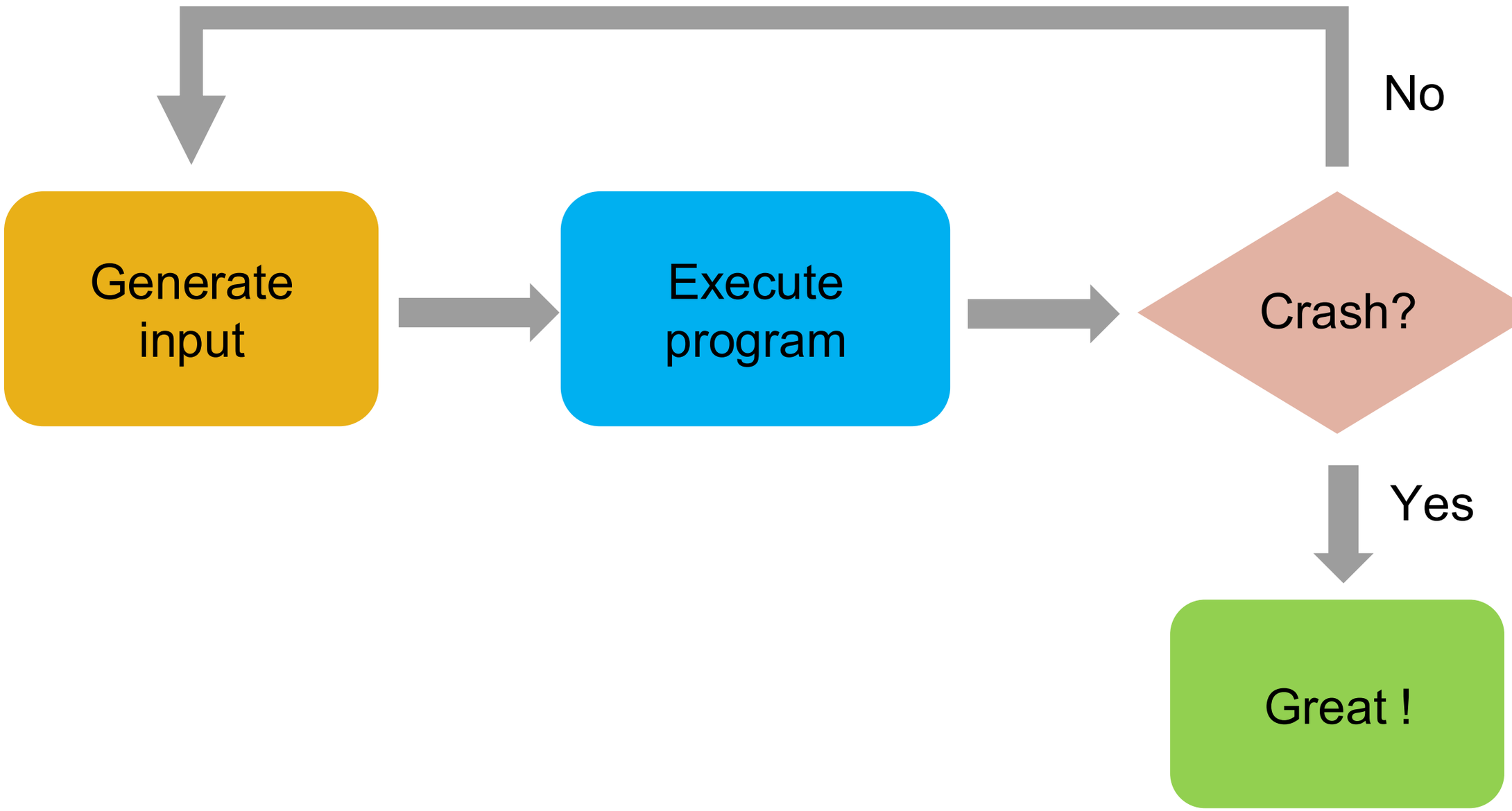
1988 Birth of the term "fuzzing" (Miller)

Monkey testing

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Fuzzing

- Feeding in random inputs until the program crashes



Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Fuzzing

- Feeding in random inputs until the program crashes

— How do we inject inputs?

— How do we generate inputs?

— How do we automate the process?

— How do we execute the program?

— How do we detect bugs?

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Fuzzing

- Feeding in random inputs until the XML parser crashes

— How do we inject inputs?

— How do we generate inputs?

— How do we detect bugs?

— How do we automate the process?

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Fuzzing

- Feeding in random inputs until the XML parser crashes

— **How do we inject inputs?** Execute the parser with an XML file

— How do we generate inputs?

— How do we detect bugs?

— How do we automate the process?

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Fuzzing

- Feeding in random inputs until the XML parser crashes

How do we inject inputs? — Execute the parser with an XML file

How do we generate inputs?

How do we detect bugs?

How do we automate the process?

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Generating Inputs for Programs

- In case of an XML parser

Idea **#1**: just generate random binary data

```
cat /dev/urandom | xml_parser
```

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Random Inputs

```
if (input[0] == '<')
    if (input[1] == 'x')
        if (input[2] == 'm')
            if (input[3] == 'l')
                // start process file
```

- Parser expects the file to start with <xml header
- We need ~2^32 guesses to get past the header check
- works poorly & is incomplete

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Generating Better Inputs for Programs

Idea #2: Model what the application should process

Structured inputs (a.k.a. structure-aware fuzzing)

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

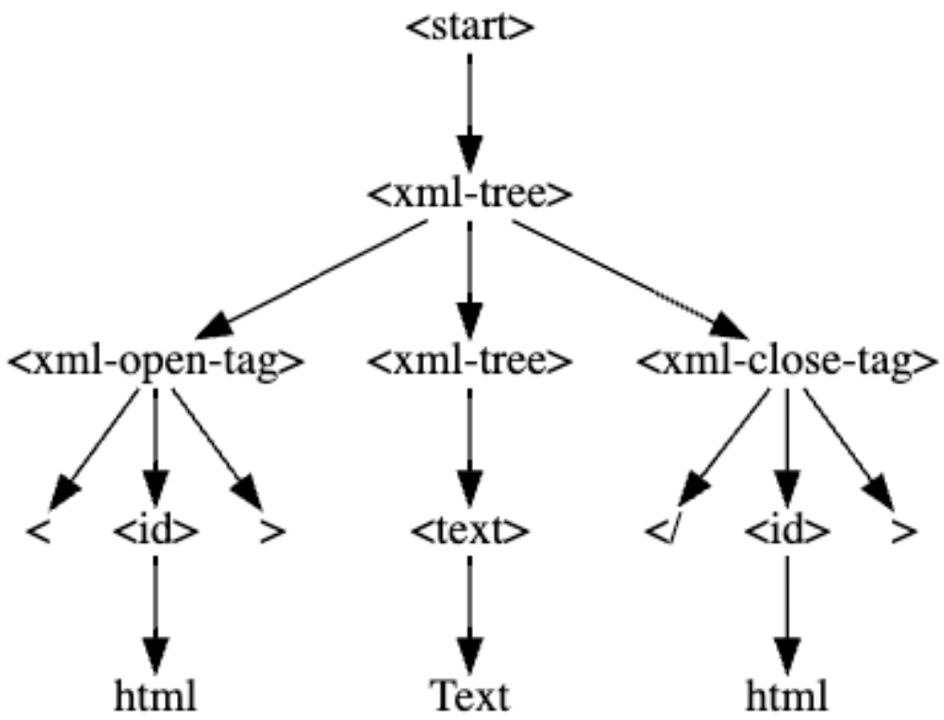
Structured Inputs

```
XML_GRAMMAR: Grammar = {
    "<start>": ["<xml-tree>"],
    "<xml-tree>": ["<text>",
                   "<xml-open-tag><xml-tree><xml-close-tag>",
                   "<xml-openclose-tag>",
                   "<xml-tree><xml-tree>"],
    "<xml-open-tag>":       ["<<id>>", "<<id> <xml-attribute>>"],
    "<xml-openclose-tag>": ["<<id>/>", "<<id> <xml-attribute>/>"],
    "<xml-close-tag>":      ["</<id>>"],
    "<xml-attribute>":      ["<id>=<id>", "<xml-attribute> <xml-attribute>"],
    "<id>":                 ["<letter>", "<id><letter>"],
    "<text>":               ["<text><letter_space>", "<letter_space>"],
    "<letter>":             srange(string.ascii_letters + string.digits +
                                   "\"" + "'" + "."),
    "<letter_space>":       srange(string.ascii_letters + string.digits +
                                   "\"" + "'" + " " + "\t"),
}
```

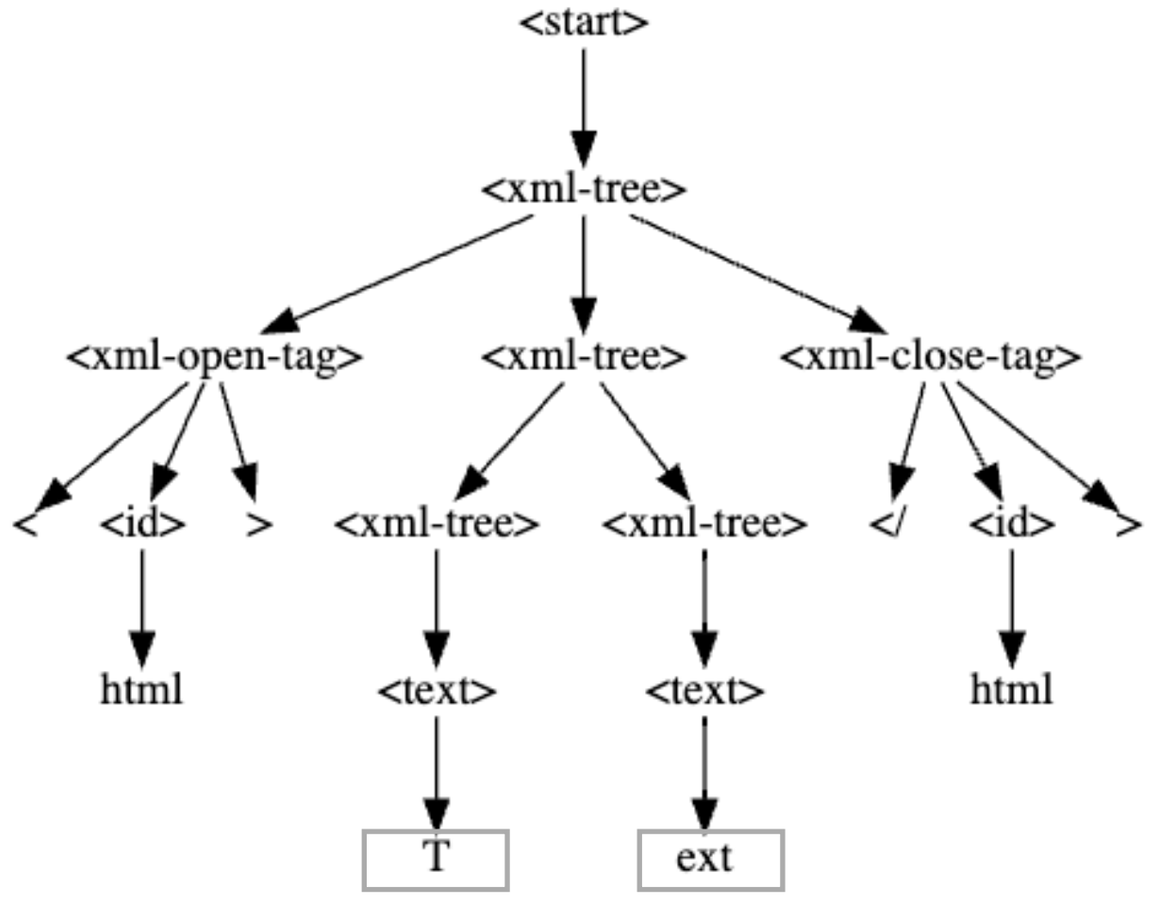
ref: https://www.fuzzingbook.org/html/GreyboxGrammarFuzzer.html#Parsing-and-Recombining-HTML

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Structured Inputs



Initial structured tree

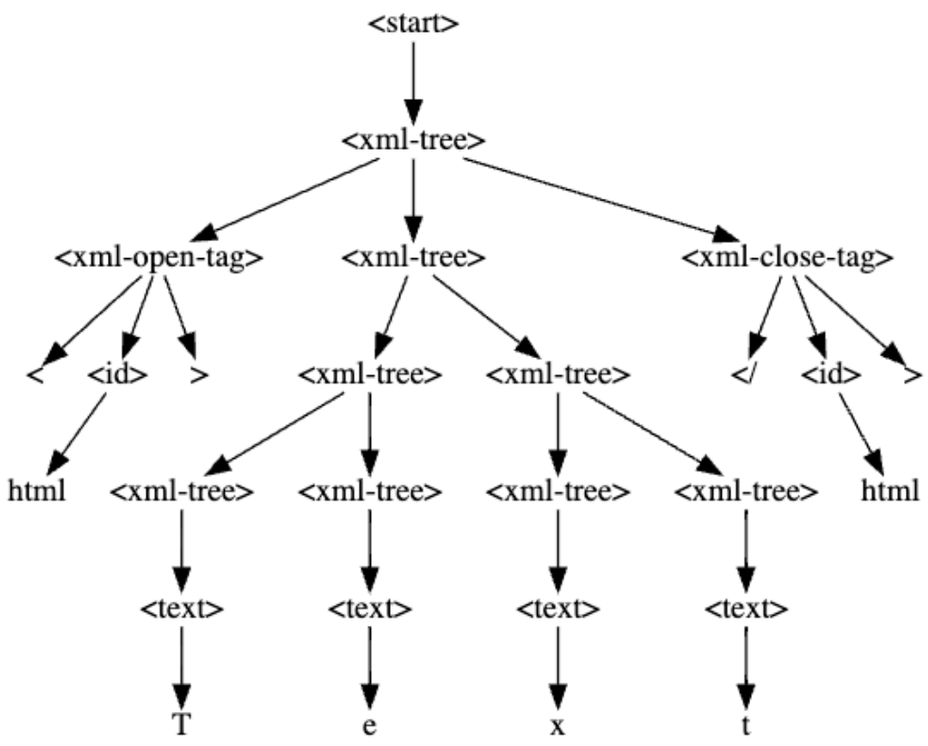


structured tree after adding nodes

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Structured Inputs

After several mutations



new structured tree

```
<start>
|-<html><head><title>Hello</title></head><body>World<br/></body></html>
<xml-tree>
|-<html><head><title>Hello</title></head><body>World<br/></body></html>
|-<head><title>Hello</title></head><body>World<br/></body>
|-<head><title>Hello</title></head>
|-<title>Hello</title>
|-Hello
|-<body>World<br/></body>
|-World<br/>
|-World
|-<br/>
<xml-open-tag>
|-<html>
|-<head>
|-<title>
|-<body>
<xml-openclose-tag>
|-<br/>
<xml-close-tag>
|-</title>
|-</head>
|-</body>
|-</html>
<xml-attribute>
<id>
<text>
<letter>
<letter_space>
```

generated testcase

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Generating Better Inputs for Programs

Idea **#3**: Coverage as completeness metric

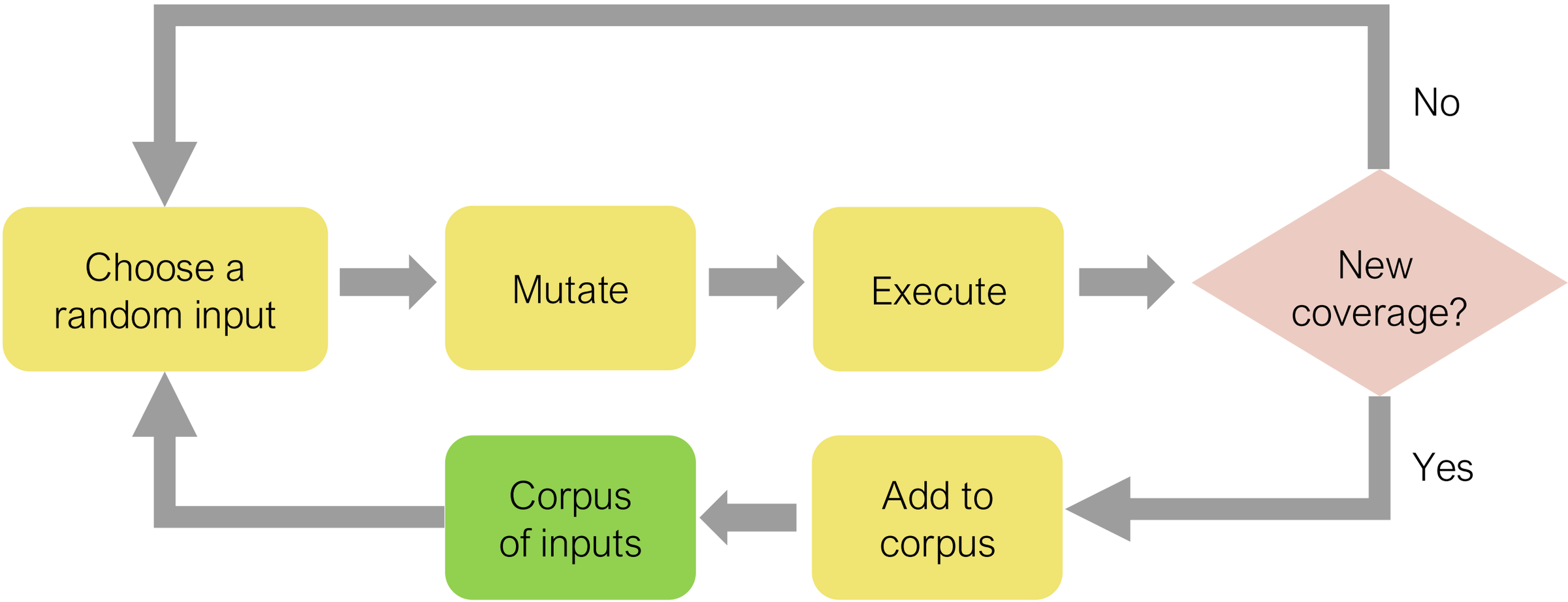
Intuition:

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

Coverage-guided generation (a.k.a. coverage-guided fuzzing)

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Coverage-Guided Generation



Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Branch Coverage

```
int arr[6];

int func(int a, int b) {
    int idx = 6;
    if (a < 6) idx -= 6; else idx -= 1;
    if (b < 6) idx -= 1; else idx += 1;
    return arr[idx]; // idx = 4
}
```

Test input: a=10, b=1

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Branch Coverage

```
int arr[6];

int func(int a, int b) {
    int idx = 6;
    if (a < 6) idx -= 6; else idx -= 1;
    if (b < 6) idx -= 1; else idx += 1;
    return arr[idx]; // idx = 1
}
```

Test input: a=1, b=10

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Branch Coverage

```
int arr[6];

int func(int a, int b) {
    int idx = 6;
    if (a < 6) idx -= 6; else idx -= 1;
    if (b < 6) idx -= 1; else idx += 1;
    return arr[idx];
}
```

All test inputs: a=10, b=1 and a=1, b=10

→ Full branch coverage

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

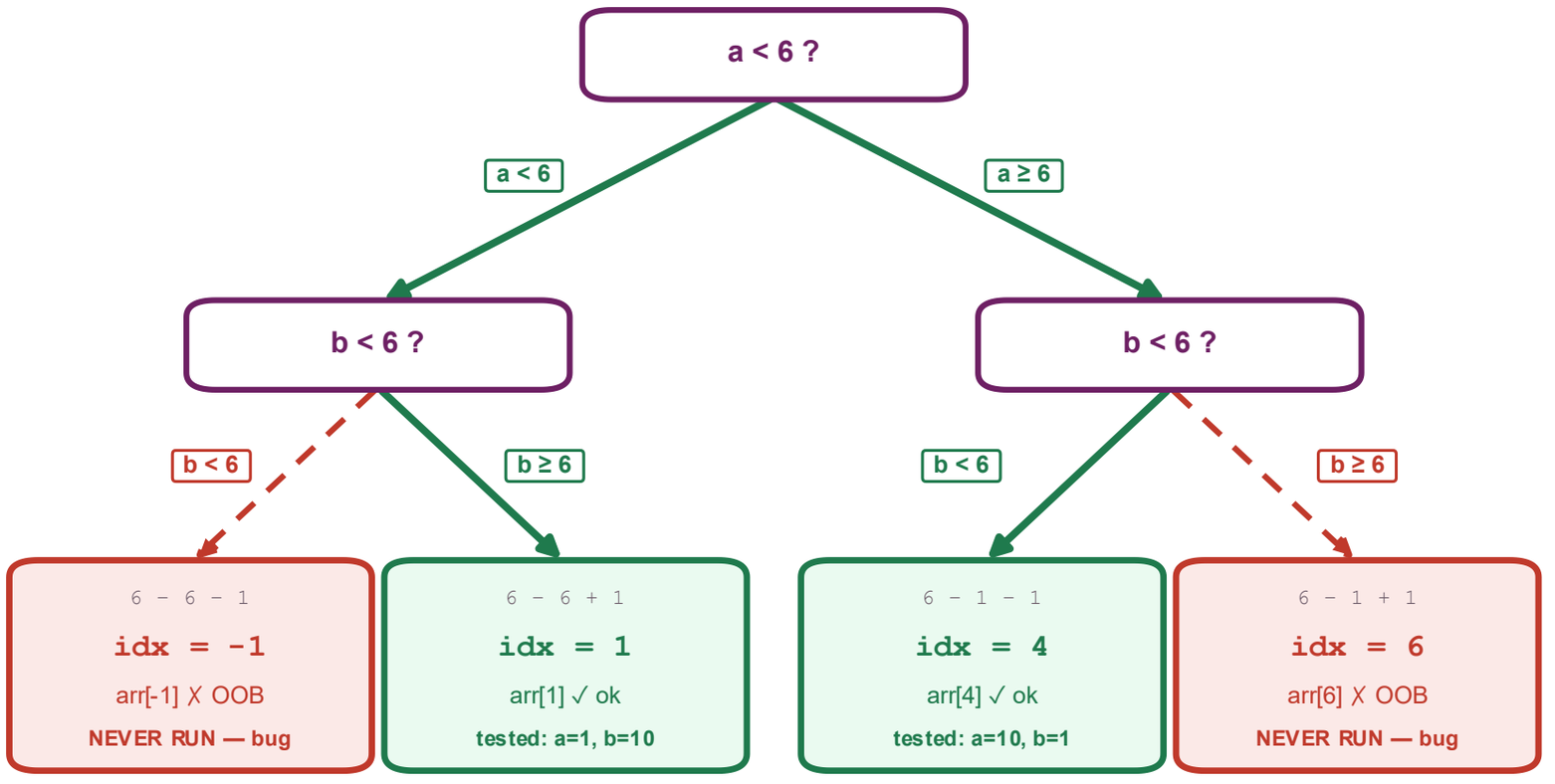
Is Branch Coverage Enough?

Branch coverage ≠ path coverage

all four branch edges are taken — yet 2 of the 4 code paths never run

```
int arr[6]; idx = 6; if (a<6) idx-=6; else idx-=1; if (b<6) idx-=1; else idx+=1; return arr[idx];
```

start: idx = 6



Coverage

taken by a test

never taken

Branch coverage

a<6 ✓ a≥6 ✓

b<6 ✓ b≥6 ✓

= 100%

Path coverage

2 / 4 paths

missed: idx = −1 and idx = 6 (both out-of-bounds)

Every decision outcome is exercised (100% branch coverage), but only 2 of the 4 root-to-leaf paths run — the both-true and both-false paths are never taken, and both are out-of-bounds.

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Is Branch Coverage Enough?

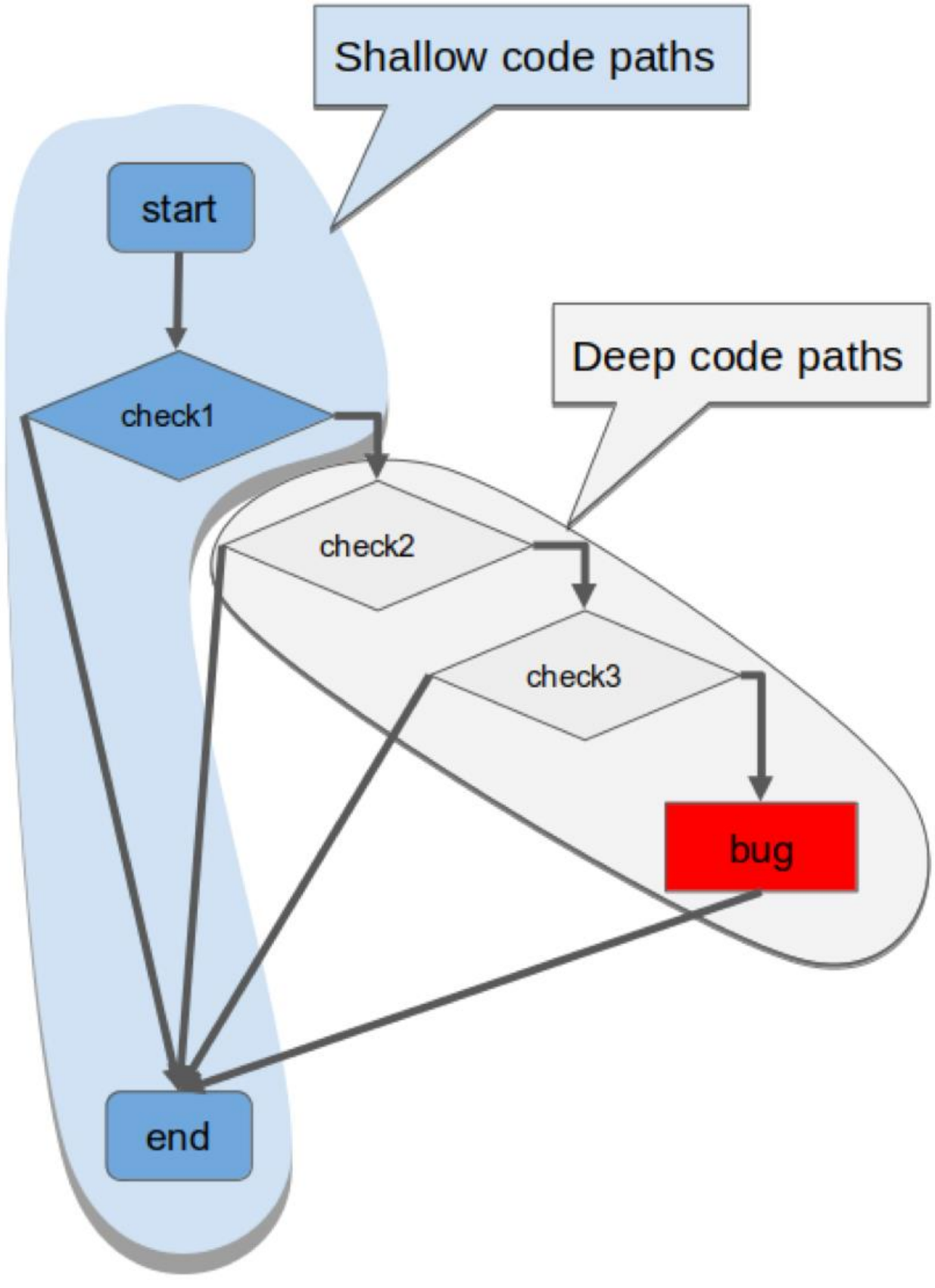
```
int arr[6];

int func(int a, int b) {
    int idx = 6;
    if (a < 6) idx -= 6; else idx -= 1;
    if (b < 6) idx -= 1; else idx += 1;
    return arr[idx];
}
```

- Not all paths are executed: Fail to find overflow bug (a = 1, b = 1)
- Full path coverage evaluates all possible paths
  - expensive (path explosion due to each branch)
  - impossible for loops
- Probabilistically covers state space

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Coverage Wall

- Hard to satisfy checks (e.g., checksum)
- Chains of checks
- Fuzzer no longer makes progress after certain iterations.



ref: CS412 Software Security

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

How to Measure Code Coverage

Existing tools:

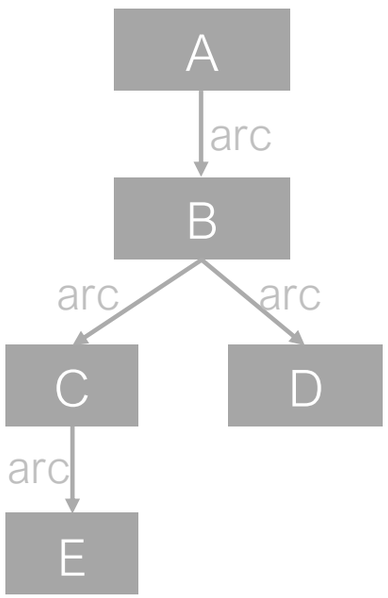
- Gcov:
  - https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
- SanitizerCoverage:
  - https://clang.llvm.org/docs/ SourceBasedCodeCoverage.html

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Example

- Compile program with coverage instrumentation
  - gcc -fprofile-arcs -ftest-coverage gcov_example.c -o example
- Generate coverage information on the fly
  - ./example
- Convert coverage information to html report
  - gcovr --html-detailed coverage.html

```c
#include <stdio.h>
int arr[5] = { 0, 1, 2, 3, 4 };

int func(int a) {
    int idx = 5;
    if (a < 5)
        idx -= 5;
    else
        idx -= 1;
    return arr[idx];
}

int main(int argc, char *argv[]){
    if (argc >= 2){
        printf("argc>=2\n");
        return 0;
    }
    printf("gcov testing.\n");
    int result = func(argc);
    return result;
}
```



arc: possible branches taken from one basic block to another

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Example

- Generated report:
  - **Green** line means executed
  - **Red** line means never executed

GCC Code Coverage Report

Directory: .
File: gcov_example.c
Date: 2022-04-14 17:13:20

	Exec	Total	Coverage
Lines:	10	13	76.9%
Functions:	2	2	100.0%
Branches:	2	4	50.0%

▸ List of functions

Line	Branch	Exec	Source
1			`#include <stdio.h>`
2			`int arr[5] = { 0, 1, 2, 3, 4 };`
3			
4		1	`int func(int a) {`
5		1	`    int idx = 5;`
6	▸ 1/2	1	`    if (a < 5)`
7		1	`        idx -= 5;`
8			`    else`
9		✗	`        idx -= 1;`
10		1	`    return arr[idx];`
11			`}`
12			
13		1	`int main(int argc, char *argv[]){`
14	▸ 1/2	1	`    if (argc >= 2){`
15		✗	`        printf("argc>=2\n");`
16		✗	`        return 0;`
17			`    }`
18		1	`    printf("gcov testing.\n");`
19		1	`    int result = func(argc);`
20		1	`    return result;`
21			`}`
22			

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Fuzzing

- Feeding in random inputs until the XML parser crashes

How do we inject inputs? Execute the parser with an XML file

How do we generate inputs? coverage-guided, mutation

How do we detect bugs?

How do we automate the process?

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Detecting Bugs

- Program crash is not a good indicator
  - Some bugs do not cause crash immediately (e.g., memory corruptions)
  - Other bugs are not crashes (e.g., uninitialized value usage)
- Use dynamic bug detectors
  - Sanitizers

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Sanitizers

- Tools based on compiler instrumentation.
- Discover bugs like **integer overflow**, **heap buffer overflow**, **use after free**, etc.
- Different type of Sanitizers:
  - ASAN
  - UBSAN
  - MSAN
  - TSAN

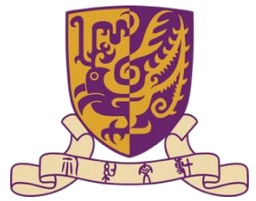
Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Address Sanitizer (ASAN)

- Fast memory error detector
- Compiler directive: **-fsanitize=address**
- Detects various issues:
  - **Out-of-bounds access to heap, stack and globals**
  - **Use After Free**
  - **Use after scope**
- Typical slowdown: ~2x

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Address Sanitizer (ASAN)



```
1    int main(int argc, char **argv) {
2      int *array = new int[100];
3      delete [] array;
4      return array[argc];  // use after free here
5    }
```

```
# compile, link, run

clang++ -g -fsanitize=address example_uaf.cc && ./a.out
```

```
=================================================================
==38960==ERROR: AddressSanitizer: heap-use-after-free on address 0x00010623a844 at pc 0x000104463f4c bp 0x00016b99f0e0 sp 0x00016b99f0d8
READ of size 4 at 0x00010623a844 thread T0
    #0 0x104463f48 in main example_uaf.cc:4
    #1 0x1048210f0 in start+0x204 (dyld:arm64+0x50f0)
    #2 0x4d7dfffffffffffc  (<unknown module>)

0x00010623a844 is located 4 bytes inside of 400-byte region [0x00010623a840,0x00010623a9d0)
freed by thread T0 here:
    #0 0x104913c70 in wrap__ZdaPv+0x6c (libclang_rt.asan_osx_dynamic.dylib:arm64+0x4bc70)
    #1 0x104463efc in main example_uaf.cc:3
    #2 0x1048210f0 in start+0x204 (dyld:arm64+0x50f0)
    #3 0x4d7dfffffffffffc  (<unknown module>)

previously allocated by thread T0 here:
    #0 0x10491387c in wrap__Znam+0x6c (libclang_rt.asan_osx_dynamic.dylib:arm64+0x4b87c)
    #1 0x104463ee4 in main example_uaf.cc:2
    #2 0x1048210f0 in start+0x204 (dyld:arm64+0x50f0)
    #3 0x4d7dfffffffffffc  (<unknown module>)

SUMMARY: AddressSanitizer: heap-use-after-free example_uaf.cc:4 in main
```

read freed buffer in `return array[argc];`

free buffer in `delete [] array;`

`int *array = new int[100];`

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Undefined Behavior Sanitizer (UBSAN)

- Compiler directive: **-fsanitize=undefined**
- Detects undefined behavior:
  - **Divide by zero**

```
int b = 0;
int c = a / b;
```

- **Signed integer overflow**

```
int k = 0x7fffffff;
k += 1;
// 0x7fffffff + 1 cannot be represented in type 'int'
```

- **Dereferencing misaligned/null pointer**

```
char * ptr = (char*)alloc_mem(LARGE_SIZE); // possible NULL pointer
*ptr = 1;
```

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Memory Sanitizer (MSAN)

- Compiler directive: **-fsanitize=memory**
- Detects uninitialized reads

```
1   int main(int argc, char** argv) {
2     int array[10]; // uninitialized stack array
3     return array[5];
4   }
```

clang -fsanitize=memory msan_example.c

```
==108578==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x4983bd in main /home/happy/workspace/msan_example.c:3:3
    #1 0x7f93e4ca00b2 in __libc_start_main /build/glibc-sMfBJT/glibc-2.31/csu/../csu/libc-start.c:308:16
    #2 0x41c22d in _start (/home/happy/workspace/a.out+0x41c22d)

SUMMARY: MemorySanitizer: use-of-uninitialized-value /home/happy/workspace/msan_example.c:3:3 in main
```

- Typical slowdown: ~3x

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Fuzzing

- Feeding in random inputs until the XML parser crashes

How do we inject inputs? Execute the parser with an XML file

How do we generate inputs? coverage-guided, mutation

How do we detect bugs? ASAN, UBSAN, MSAN, etc.

How do we automate the process?

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Automation

- Run program with a generated testcase
- Monitor program for coverage and crashes
- Deduplicate crashes
- Minimize testcase
- Generate reproducers
- Report crashes

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

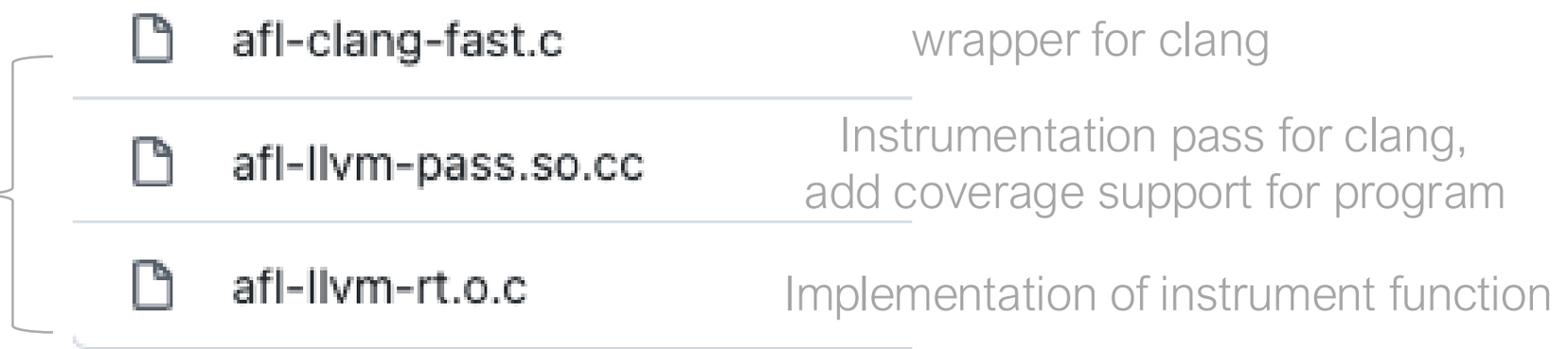
AFL

- The most well-known fuzzer currently
- Uses compiler-time instrumentation to track branch coverage
- Mutate fuzzing testcase based on previous branch coverage

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Hands On: Building AFL

- git clone https://github.com/google/AFL.git
- make
- cd llvm_mode
- make



- **afl-clang-fast**: compiler which instruments program while building
- **afl-fuzz**: overall fuzzer

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Hands On: Fuzzing Example

```c
#include <stdio.h>
#include <unistd.h>

#define SIZE 20

int main(int argc, char *argv[])
{
    char input[SIZE] = {0};
    size_t length;
    length = read(STDIN_FILENO, input, SIZE);
    char calc_value = 0;

    if (input[0] == 'a'){
        if (input[1] == 'b'){
            if (input[2] == 'c'){
                int idx = input[3] + input[4];
                calc_value = input[idx];
            }
        }
    }

    return calc_value;
}
```

buffer overflow bug

- Example PoC:

  input buffer: “abc\x40\x40"

Credits: based on Andrey Konovalov’s YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Hands On: Fuzzing Example

- Build program with ASan & Coverage instrumentation
  - afl-clang-fast -fsanitize=address example.c
- Provide initial corpus/testcase seed
  - echo 123 > corpus/seeds0
- Fuzz it
  - afl-fuzz -D -i input -o output -- ./a.out
    - -D: enable deterministic fuzzing (more mutation strategies)
    - -i dir: input directory with initial testcases.
    - -o dir: output directory for fuzzer findings

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Hands On: Fuzzing Example

One cycle:

- go over all the interesting test cases discovered so far

```
                    american fuzzy lop 2.57b (a.out)
┌─ process timing ─────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 3 sec │  cycles done : 5      │
│   last new path : 0 days, 0 hrs, 0 min, 1 sec │  total paths : 4      │
│ last uniq crash : 0 days, 0 hrs, 0 min, 2 sec │ uniq crashes : 1      │
│  last uniq hang : none seen yet               │   uniq hangs : 0      │
├─ cycle progress ─────┬─ map coverage ─────────┴───────────────┘
│  now processing : 3 (75.00%)  │    map density : 0.01% / 0.01%
│ paths timed out : 0 (0.00%)   │ count coverage : 1.00 bits/tuple
├─ stage progress ─────┼─ findings in depth ────────────────────┐
│  now trying : havoc           │ favored paths : 1 (25.00%)    │
│ stage execs : 4888/12.3k (39.78%) │  new edges on : 4 (100.00%)   │
│ total execs : 19.7k           │ total crashes : 2 (1 unique)  │
│  exec speed : 5237/sec        │  total tmouts : 0 (0 unique)  │
├─ fuzzing strategy yields ────┴─ path geometry ──────────────┤
│   bit flips : 0/120, 1/116, 1/108  │    levels : 4                 │
│  byte flips : 0/15, 0/11, 0/3      │   pending : 1                 │
│ arithmetics : 0/840, 0/160, 0/0    │  pend fav : 1                 │
│  known ints : 0/70, 0/298, 0/132   │ own finds : 3                 │
│  dictionary : 0/0, 0/0, 0/0        │  imported : n/a               │
│       havoc : 2/9152, 0/3792       │ stability : 100.00%           │
│        trim : 50.00%/1, 0.00%      ├───────────────────────────────┘
└────────────────────────────────────┘                 [cpu000: 16%]
```

branch tuples already hit/the bitmap can hold

the number of crashes

consistency of observed traces

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Hands On: Fuzzing Example

Output (fuzzer status & findings):

```
output
├── crashes
│   ├── id:000000,sig:06,src:000002,op:flip4,pos:2
│   └── README.txt
├── fuzz_bitmap
├── fuzzer_stats
├── hangs
├── plot_data
└── queue
    ├── id:000000,orig:seeds0
    ├── id:000001,src:000000,op:havoc,rep:64,+cov
    ├── id:000002,src:000001,op:flip2,pos:1,+cov
    └── id:000003,src:000002,op:havoc,rep:4,+cov

3 directories, 9 files
```

testcase which crashes the program

fuzzing status. For clustered fuzzers, use **afl-whatsup** to get status of fuzzers

interesting testcases discovered

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Hands On: Fuzzing Example

Reproduce crashes:

```
% hexdump -C output/crashes/id:000000,sig:06,src:000002,op:flip4,pos:2
00000000  61 62 63 ff                                       |abc.|
```

"abc\xff"

```
$ cat output/crashes/id:000000,sig:06,src:000002,op:flip4,pos:2| ./a.out
=================================================================
==4016427==ERROR: AddressSanitizer: stack-buffer-underflow on address 0x7ffee926c62f at pc 0x555c95a4e729 bp 0x7ffee926c600 sp 0x7ffee926c5f0
READ of size 1 at 0x7ffee926c62f thread T0
    #0 0x555c95a4e728 in main /home/happy/fuzz/afl_lab/example.c:17
    #1 0x7ff1904090b2 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x240b2)
    #2 0x555c95a4e7ad in _start (/home/happy/fuzz/afl_lab/a.out+0x17ad)

Address 0x7ffee926c62f is located in stack of thread T0 at offset 31 in frame
    #0 0x555c95a4e22f in main /home/happy/fuzz/afl_lab/example.c:7

  This frame has 1 object(s):
    [32, 52) 'input' (line 8) <== Memory access at offset 31 underflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
```

Credits: based on Andrey Konovalov's YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

Conclusion

Three Shades of Fuzzing

Black box

- Unaware of the program structure
- Scale but dumb

White box

- Leverage semantic program analysis
- Smart but don’t scale too much

Grey box

- Leverages program instrumentation
- Smart & scale

Credits: based on Andrey Konovalov’s YouTube video https://www.youtube.com/watch?v=HN2qFplvATk

香港中文大學
CUHK

Exploit Mitigations

Exploit Mitigations

- Scenario
  - Someone has just handed you a large, existing codebase
  - It's not written in a memory-safe language, and it wasn't written with memory safety in mind
  - How can you protect this code from exploits without having to completely rewrite it?

Exploit Mitigations

- **Exploit mitigations (code hardening)**: Compiler and runtime defenses that make common exploits harder
  - Find ways to turn attempted exploits into program crashes
  - Crashing is safer than exploitation: The attacker can crash our system, but at least they can't execute arbitrary code
  - Mitigations are cheap (low overhead) but not free (some costs associated with them)

Recall: Putting Together an Attack

- Find a memory safety (e.g. buffer overflow) vulnerability
- Write malicious shellcode at a known memory address
- Overwrite the RIP with the address of the shellcode
- Return from the function
- Begin executing malicious shellcode

Recall: Putting Together an Attack

- Find a memory safety (e.g. buffer overflow) vulnerability
- **Write malicious shellcode at a known memory address**
- **Overwrite the RIP with the address of the shellcode**
- Return from the function
- **Begin executing malicious shellcode**

- We can defend against memory safety vulnerabilities by making each of these steps more difficult (or impossible)!

香港中文大學
CUHK

Mitigation: Non-Executable Pages

Recall: Putting Together an Attack

- Find a memory safety (e.g. buffer overflow) vulnerability
- **Write malicious shellcode at a known memory address**
- **Overwrite the RIP with the address of the shellcode**
- Return from the function
- **Begin executing malicious shellcode**
  - Mitigation: Non-executable pages
- We can defend against memory safety vulnerabilities by making each of these steps more difficult (or impossible)!

Non-Executable Pages

- Idea: Most programs don't need memory that is both written to and executed, so make portions of memory **either** executable **or** writable but not both
  - Stack, heap, and static data: Writable but not executable
  - Code: Executable but not writable
- Page table entries have a writable bit and an executable bit that can be set to achieve this behavior
  - Page tables convert virtual addresses to physical addresses
  - Implemented in hardware, so effectively 0 overhead!

Non-Executable Pages

- Also known as
  - **W^X** (write XOR execute)
  - **DEP** (Data Execution Prevention, name used by Windows)
  - **No-execute bit** (the name of the bit itself)

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00160 0x00160 R   0x4
  INTERP         0x000194 0x08048194 0x08048194 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x003a0 0x003a0 R   0x1000
  LOAD           0x001000 0x08049000 0x08049000 0x00288 0x00288 R E 0x1000
  LOAD           0x002000 0x0804a000 0x0804a000 0x0014c 0x0014c R   0x1000
  LOAD           0x002f00 0x0804bf00 0x0804bf00 0x00120 0x00124 RW  0x1000
  DYNAMIC        0x002f08 0x0804bf08 0x0804bf08 0x000e8 0x000e8 RW  0x4
  NOTE           0x0001a8 0x080481a8 0x080481a8 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x002024 0x0804a024 0x0804a024 0x0003c 0x0003c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
  GNU_RELRO      0x002f00 0x0804bf00 0x0804bf00 0x00100 0x00100 R   0x1
```

Subverting Non-Executable Pages

- Issue: Non-executable pages doesn't prevent an attacker from leveraging **existing** code in memory as part of the exploit
- Most programs have many functions loaded into memory that can be used for malicious behavior
- **Return-to-libc**: An exploit technique that overwrites the RIP to jump to a function in the standard C library (libc) or a common operating system function
- **Return-oriented programming (ROP)**: Constructing custom shellcode using pieces of code that already exist in memory

Subverting Non-Executable Pages: Return-to-libc

- Recall: Per the x86 calling convention, each program expects arguments to be placed directly above the RIP
- Consider the **system** function, which executes a shell command. We want to execute it like this:

```
char cmd[] = "rm -rf /";
system(cmd);
```

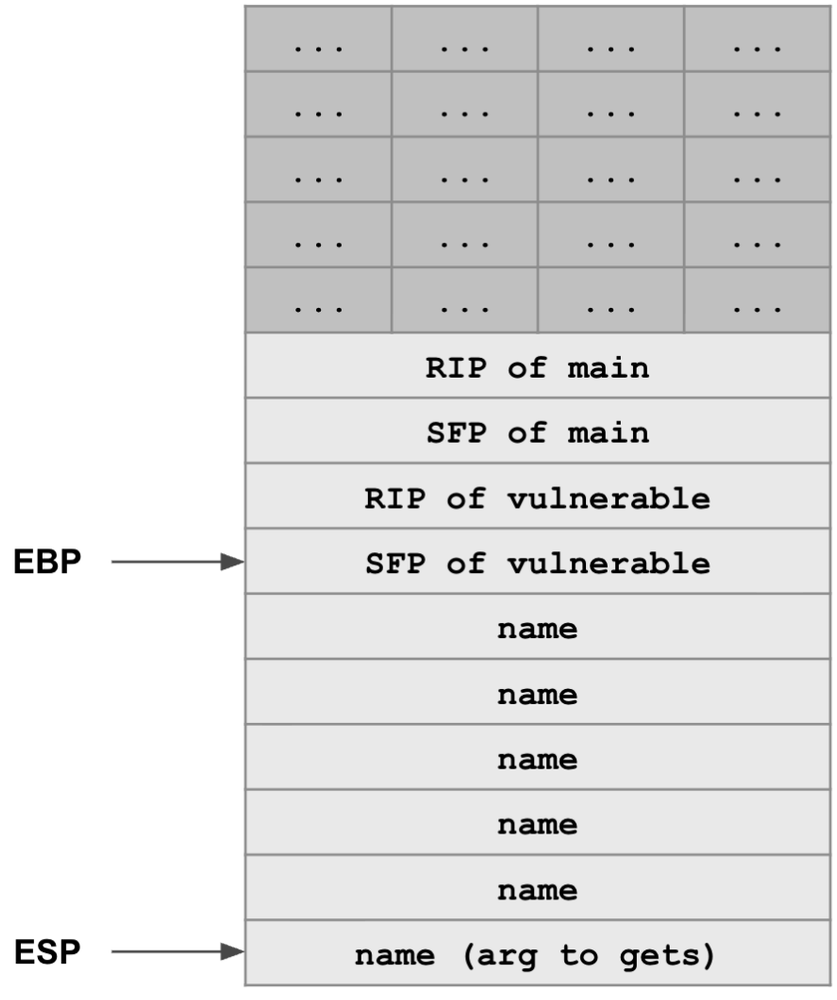
Subverting Non-Executable Pages: Return-to-libc

Exploit:

```
'A' * 24
    + [address of system]
    + 'B' * 4
    + [address of "rm -rf /"]
    + "rm -rf /"
```

```
int system(char *command);

void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
system:
    ...

vulnerable:
    ...
EIP →  call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



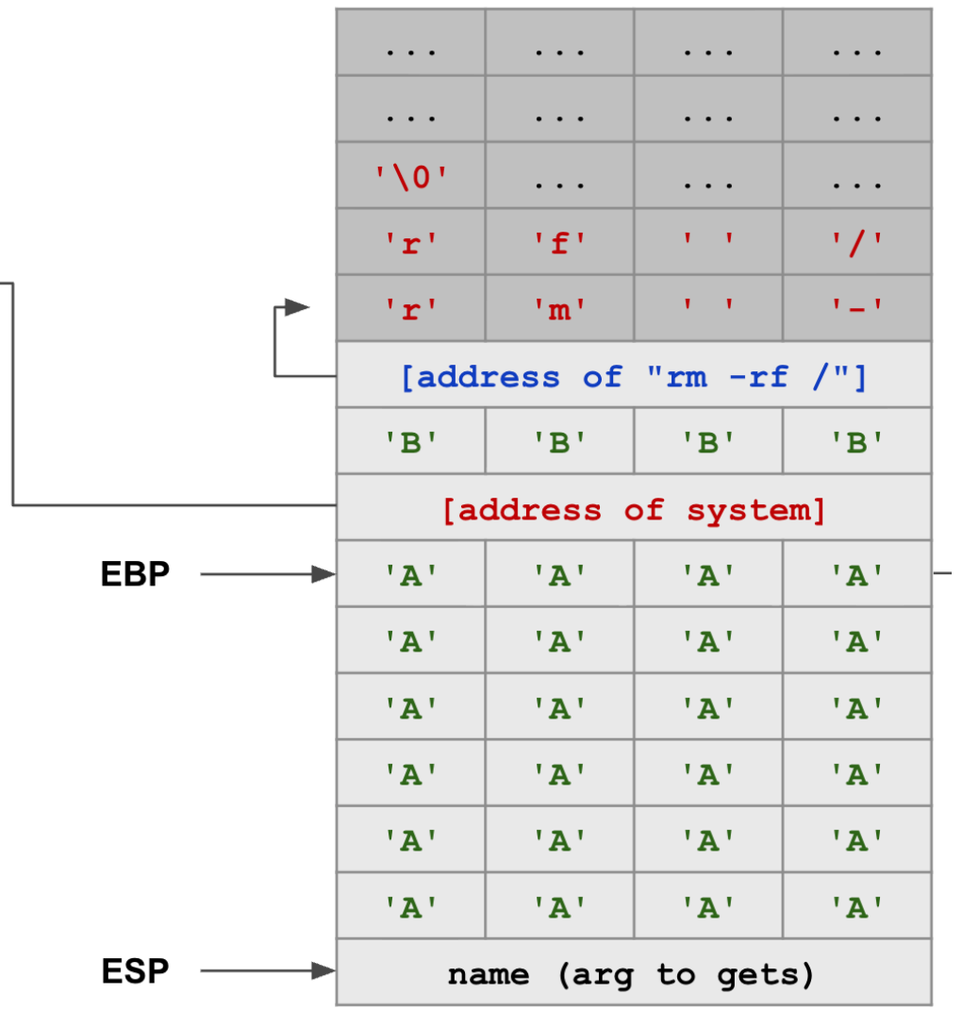
Subverting Non-Executable Pages: Return-to-libc

Exploit:

```
'A' * 24
    + [address of system]
    + 'B' * 4
    + [address of "rm -rf /"]
    + "rm -rf /"
```

```
int system(char *command);

void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
system:
    ...

vulnerable:
    ...
    call gets
EIP → addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

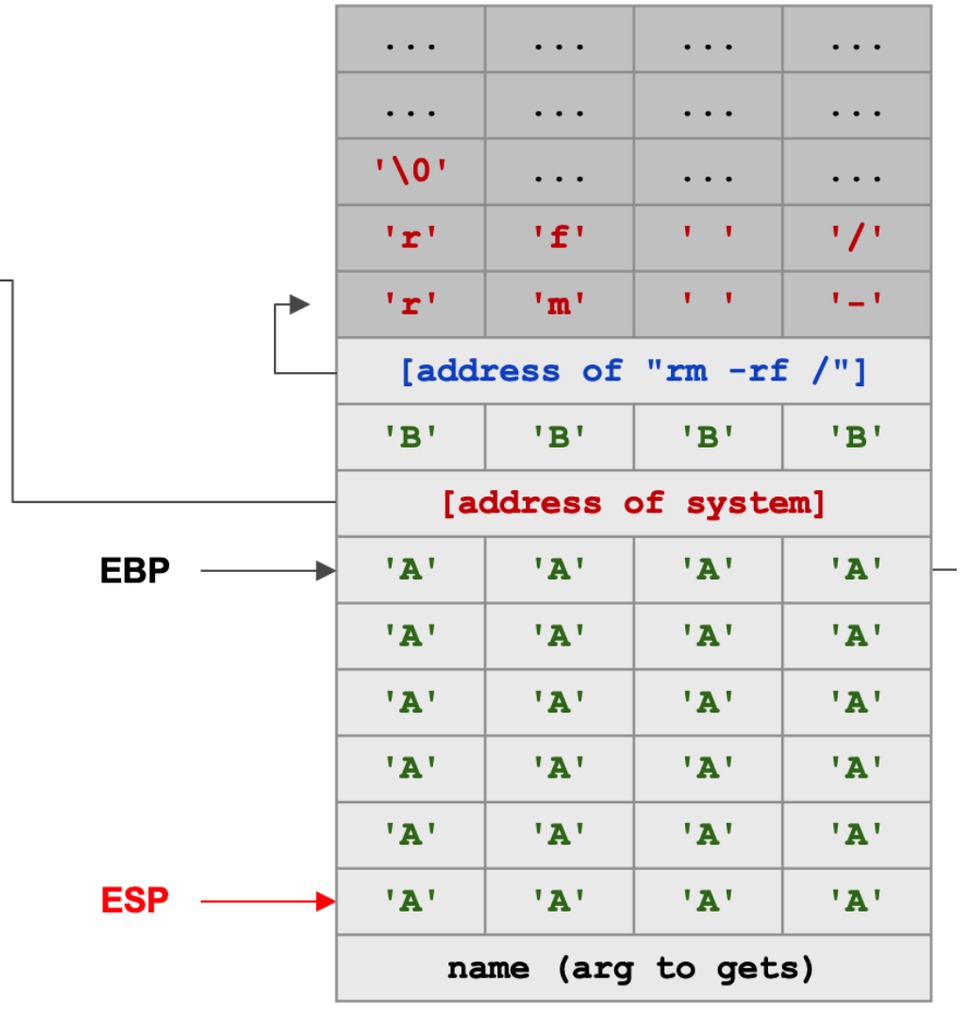


Subverting Non-Executable Pages: Return-to-libc

```
int system(char *command);

void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
system:
    ...

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

EIP →



Subverting Non-Executable Pages: Return-to-libc

```
int system(char *command);

void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
system:
    ...

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

EIP → popl %ebp

ESP → EBP

...	...	...	...
...	...	...	...
'\0'	...	...	...
'r'	'f'	' '	'/'
'r'	'm'	' '	'-'
[address of "rm -rf /"]			
'B'	'B'	'B'	'B'
[address of system]			
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
name (arg to gets)			

Subverting Non-Executable Pages: Return-to-libc

```
int system(char *command);

void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
system:
    ...

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

EIP →

...	...	...	...
...	...	...	...
'\0'	...	...	...
'r'	'f'	' '	'/'
'r'	'm'	' '	'-'
[address of "rm -rf /"]			
'B'	'B'	'B'	'B'
[address of system]			
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
name (arg to gets)			

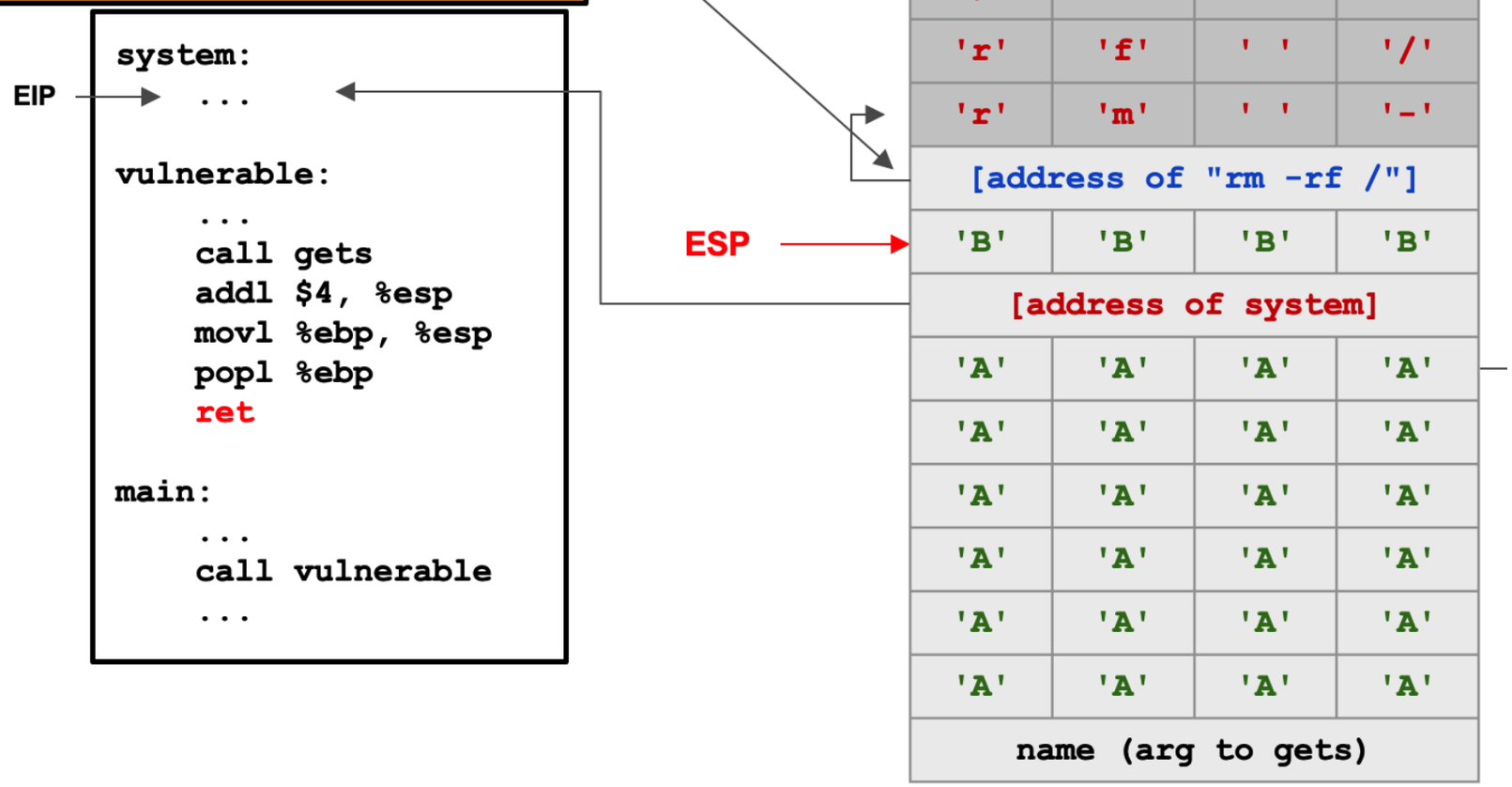
ESP →

Subverting Non-Executable Pages: Return-to-libc

We jumped into the **`system`** function, and it expects the first argument to be 4 bytes above the ESP: **`"rm -rf /"`**!

```
int system(char *command);

void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
system:
    ...

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



香港中文大學
CUHK

Mitigation: Stack Canaries

Recall: Putting Together an Attack

- Find a memory safety (e.g. buffer overflow) vulnerability
- **Write malicious shellcode at a known memory address**
- **Overwrite the RIP with the address of the shellcode**
  - Mitigation: Stack canaries
- Return from the function
- **Begin executing malicious shellcode**
- We can defend against memory safety vulnerabilities by making each of these steps more difficult (or impossible)!

Analogy: Canary in a Coal Mine

- Miners protect themselves against toxic gas buildup in the mine with a canary
  - Canary: A small, noisy bird that is sensitive to toxic gas
  - If toxic gas builds up, the canary dies first
    - The miners notice that the canary has died and leave the mine
- The canary in the coal mine is a sacrificial animal
  - The miners don’t expect the canary to survive
  - However, the canary's death is a warning sign that saves the lives of the miners

Analogy: Canary in a Coal Mine

- **Takeaway**: Let's put a sacrificial value (a canary) on the stack
  - The value is not meaningful (we don't care if it's preserved)
  - The code never uses or changes this value
  - If the value changes, that's a warning sign that somebody is messing with our code!

Stack Canaries

- Idea: Add a sacrificial value on the stack, and check if it has been changed
  - When the program runs, generate a random secret value and save it in the canary storage
  - In the function prologue, place the canary value on the stack right below the SFP/RIP
  - In the function epilogue, check the value on the stack and compare it against the value in canary storage
- The canary value is never actually used by the function, so if it changes, somebody is probably attacking our system!

Stack Canaries: Properties

- A canary value is unique every time the program runs but the same for all functions **within a run**
- Usually, the canary value is chosen so its first byte is **a null byte**, as extra hardening against string-based attacks
  - Why?
  - A string-based overflow will write a bunch of non-null bytes, followed by one null byte, and can't keep writing anything further after the first null byte, so any string-based overflow will have to either stop at the canary or change the canary

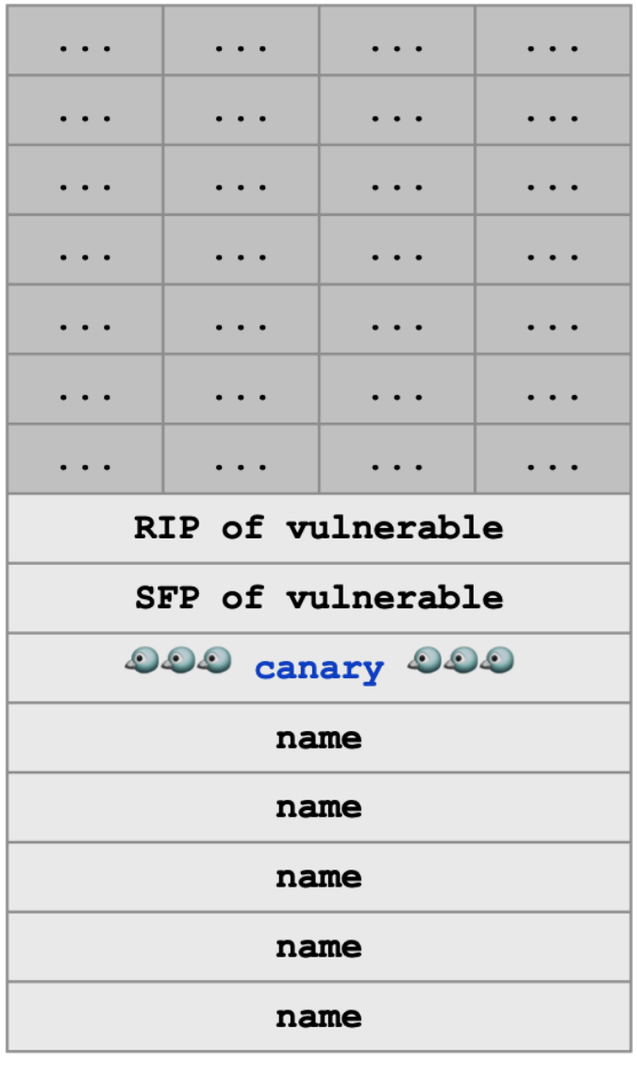
Stack Canaries

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

```
vulnerable:
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp
    movl ($CANARY_ADDR), %eax # Load canary
    movl %eax, -4(%ebp)       # Save on stack

    ...

    movl -4(%ebp), %eax       # Load stack value
    cmpl %eax, ($CANARY_ADDR) # Compare to canary and...
    jne canary_failed         # ... crash if not equal
    movl %ebp, %esp
    popl %ebp
    ret
```

Because the write starts at name, the attacker has to overwrite the canary before the RIP or SFP!

Note: 20 bytes for **name** + 4 bytes for canary (32-bit architecture)



Stack Canaries: Efficiency

- Compiler inserts a few extra instructions at start and end of each function to set/check the canary, so there is more overhead
- In almost all applications, the performance impact is insignificant

Shadow Stack

- How it works
  - A second CPU-protected stack holds only return addresses
  - Call pushes the return address to both stacks; ret makes the CPU compare them
  - Mismatch: #CP (control-protection) fault

Shadow Stack

- How it stops our attack
  - The overflow rewrites the normal saved return address – it cannot reach the shadow stack
  - On ret, tampered value != saved shadow copy -> hijack detected
- vs canary: a canary is a tripwire that can be leaked; the shadow stack enforces integrity in hardware, no secret needed.

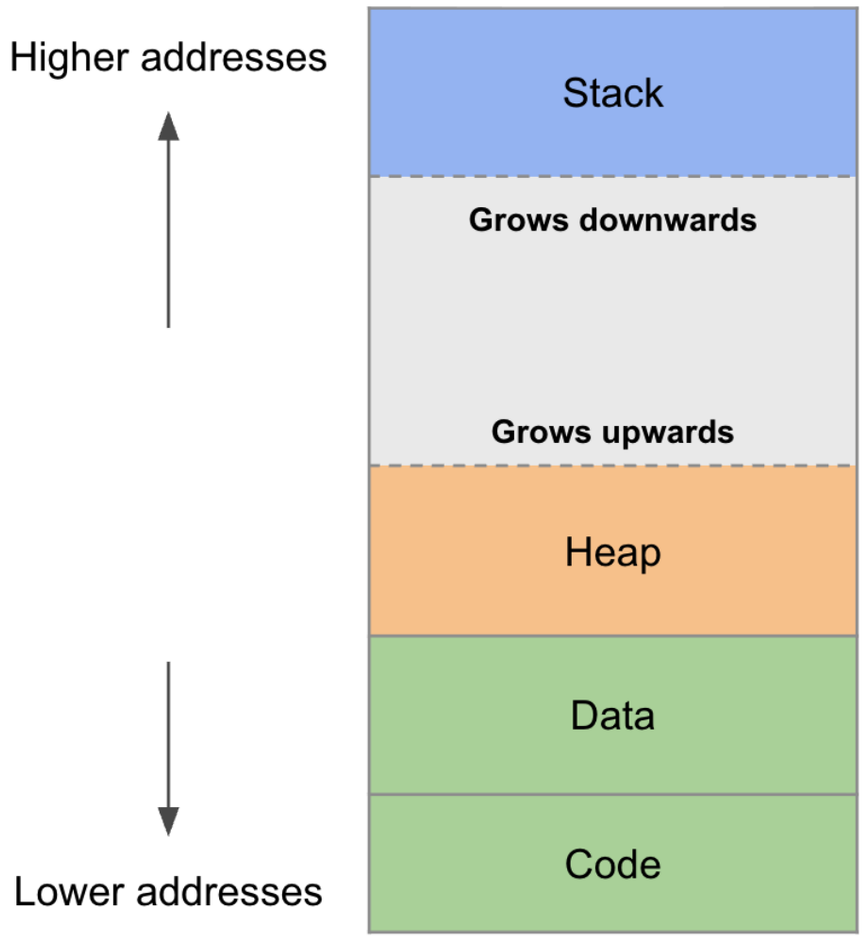
香港中文大學
CUHK

Mitigation: Address Space Layout Randomization (ASLR)

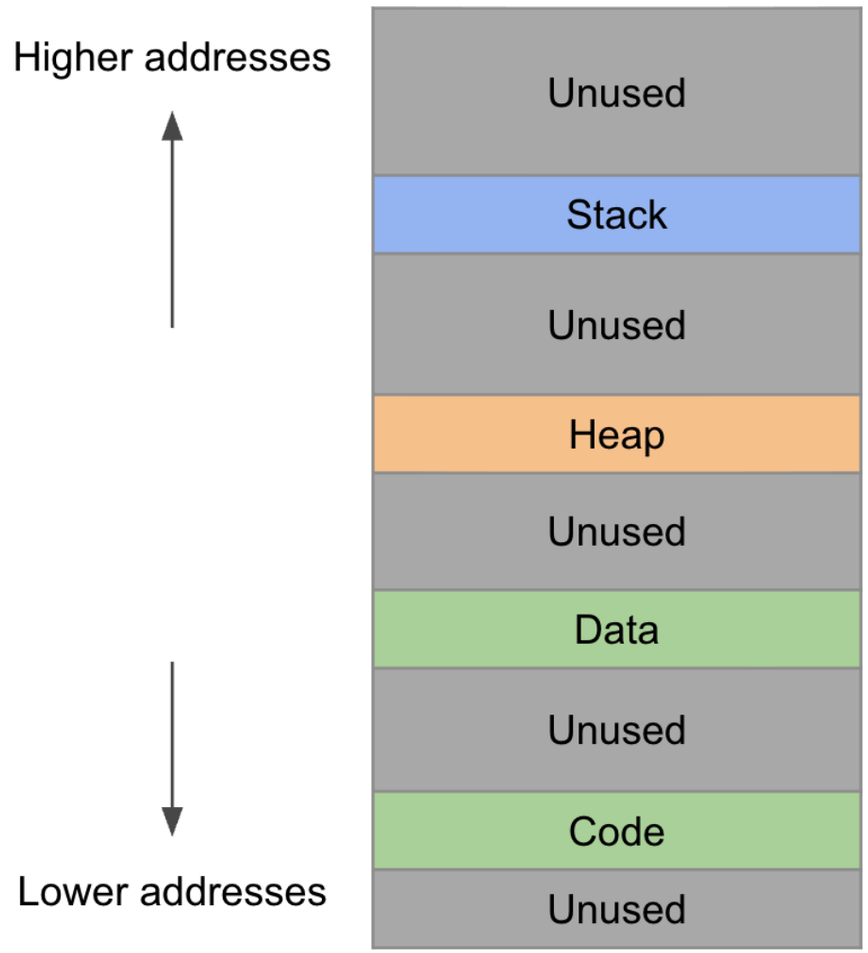
Recall: Putting Together an Attack

- Find a memory safety (e.g. buffer overflow) vulnerability
- **Write malicious shellcode at a known memory address**
  - Mitigation: Address-space layout randomization
- **Overwrite the RIP with the address of the shellcode**
- Return from the function
- **Begin executing malicious shellcode**
- We can defend against memory safety vulnerabilities by making each of these steps more difficult (or impossible)!

Recall: x86 Memory Layout

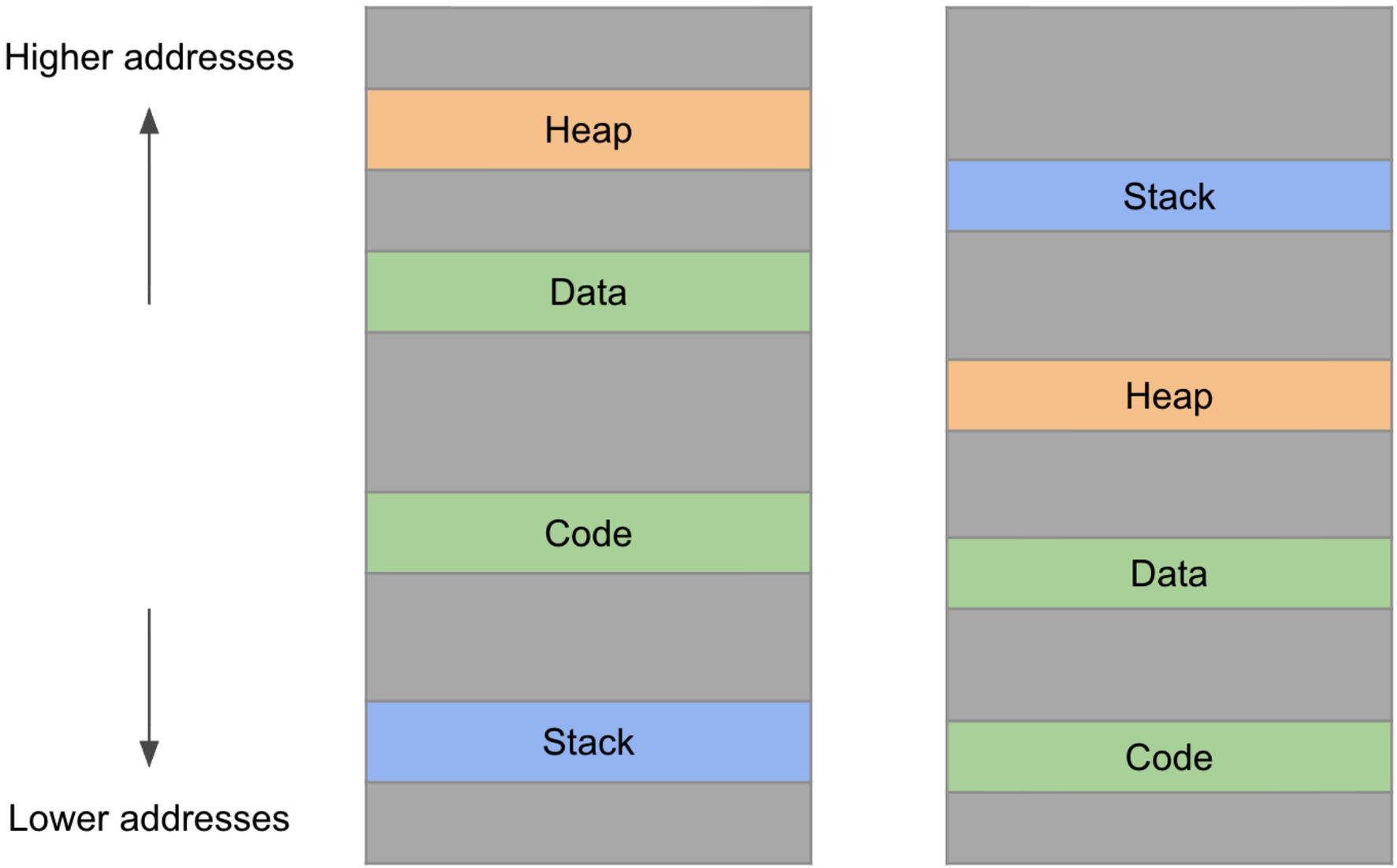


In theory, x86 memory layout looks like this...



...but in practice, it usually looks like this (mostly empty)!

Recall: x86 Memory Layout



Idea: Put each segment of memory in a different location each time the program is run

Address Space Layout Randomization

- **Address space layout randomization (ASLR)**: Put each segment of memory in a different, random location each time the program is run
  - The attacker can't know where their shellcode will be because its address changes every time you run the program

Address Space Layout Randomization

- ASLR can shuffle all four segments of memory
  - Randomize the stack: Can't place shellcode on the stack without knowing the address of the stack
  - Randomize the heap: Can't place shellcode on the heap without knowing the address of the heap
- Randomize the code: Can't construct a ROP chain or return-to-libc attack without knowing the address of code
- Within each segment of memory, relative addresses are the same (e.g. the RIP is always 4 bytes above the SFP)

ASLR: Efficiency

- ASLR has very low performance overhead
  - Programs are dynamically linked at runtime
  - We already have to do the work of going through the executable and rewriting code to contain known addresses before executing it, so not much extra cost to ASLR

Subverting ASLR

- Force the program to **reveal the address of a pointer**, whose address relative to your shellcode is known
  - Relative addresses are usually fixed, so this is sufficient to undo randomization!
  - Leak a stack pointer: Leak the location of the stack
  - Leak an RIP: Leak the location of the caller

Subverting ASLR

- Guess the address of your shellcode: Brute-force
  - Randomization usually happens on page boundaries (usually 12 bits for 4 KiB pages)
  - 32-bit: 32 - 12 = 20 bits, 2^{20} possible pages, which is feasibly brute-forced
  - 64-bit (usually 48-bit addressing): 48 - 12 = 36 bits, 2^{36} possible pages

香港中文大學
CUHK

Combining Mitigations

Combining Mitigations

- Recall: We can use multiple mitigations together
  - Synergistic protection: one mitigation helps strengthen another mitigation
  - Force the attacker to find multiple vulnerabilities to exploit the program
  - Defense in depth

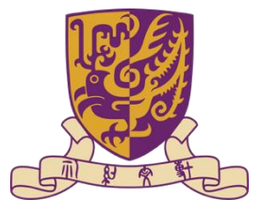
Combining Mitigations

- Example: Combining ASLR and non-executable pages
  - An attacker can't write their own shellcode, because of non-executable pages
  - An attacker can't use existing code in memory, because they don't know the addresses of those code (ASLR)

Combining Mitigations

- To defeat ASLR *and* non-executable pages, the attacker needs to find two vulnerabilities
  - First, find a way to leak memory and reveal the address randomization (defeat ASLR)
  - Second, find a way to write to memory and write a ROP chain (defeat non-executable pages)

Enabling Mitigations



- Many mitigations (stack canaries, non-executable pages, ASLR) are nearly free today (insignificant performance impact)
- The programmer sometimes has to manually enable mitigations
  - Example: Enable ASLR and non-executable pages when running a program
  - Example: Setting a flag to compile a program with stack canaries

Enabling Mitigations

- If the default is that the mitigation is disabled, many programs won't change the default
  - Recall: Consider human factors!
  - Recall: Use fail-safe defaults!

Summary: Memory Safety Mitigations

- Memory-safe languages
  - Using a memory-safe language (e.g. Python, Java) stops all memory safety vulnerabilities.
  - Why use a non-memory-safe language?
    - Commonly-cited reason, but mostly a myth: Performance
    - Real reason: Legacy, existing code

Summary: Memory Safety Mitigations

- Writing memory-safe code
  - Carefully write and reason about your code to ensure memory safety in a non-memory-safe language
  - Requires programmer discipline, and can be tedious sometimes

Summary: Memory Safety Mitigations

- Building secure software
  - Use tools for analyzing and patching insecure code
  - Test your code for memory safety vulnerabilities
    - Fuzz testing
- Keep any external libraries updated for security patches

Summary: Memory Safety Mitigations

- Mitigation: **Non-executable pages**
  - Make portions of memory either executable or writable, but not both
  - Defeats attacker writing shellcode to memory and executing it
- Subversions
  - **Return-to-libc**: Execute an existing function in the C library
  - **Return-oriented programming** (**ROP**): Create your own code by chaining together small gadgets in existing library code (Did not discuss in the lecture)

Summary: Memory Safety Mitigations

- Mitigation: **Stack canaries**
  - Add a sacrificial value on the stack. If the canary has been changed, someone's probably attacking our system
  - Defeats attacker overwriting the RIP with address of shellcode
- Subversions
  - An attacker can write around the canary
  - The canary can be leaked by another vulnerability (e.g. format string vulnerability)
  - The canary can be brute-forced by the attacker

Summary: Memory Safety Mitigations

- Mitigation: **Address space layout randomization (ASLR)**
  - Put each segment of memory in a different location each time the program is run
  - Defeats attacker knowing the address of shellcode
- Subversions
  - Leak addresses with another vulnerability
  - Brute-force attack to guess the addresses

Summary: Memory Safety Mitigations

- Combining mitigations
  - Using multiple mitigations usually forces the attacker to find multiple vulnerabilities to exploit the program (defense-in-depth)

One warm-up, two bugs

a length check that passes does not make the code safe

```
void func(int len, char *data) {
char buf[64];
if (len > 64)                          ① signed guard
return;
memcpy(buf, data, len);                ② raw copy, no '\0'
}
```

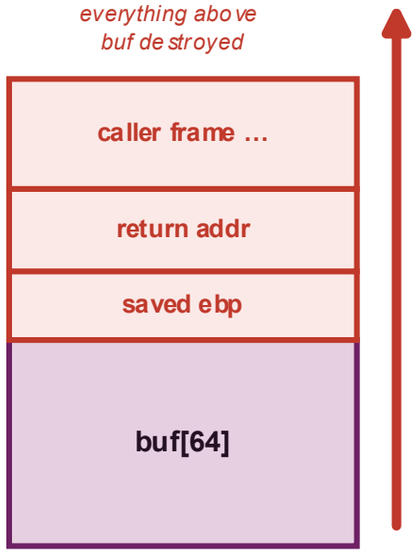
① Negative len → write overflow

len = -1 (int)

len > 64 → false
✗ check is bypassed

memcpy needs size_t:
-1 → 0xFFFF…FFFF
≈ 1.8×10¹⁹ bytes

everything above buf destroyed



OOB WRITE

Huge copy smashes the stack → crash / hijack

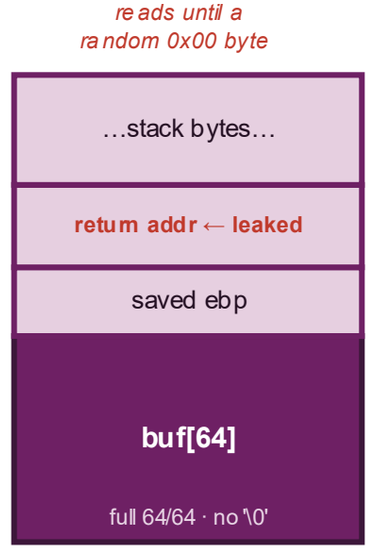
② len = 64 → buf not NUL-terminated

len = 64 ✓ passes (64 = 64)

memcpy copies 64 bytes —
no '\0' is appended

later: strlen(buf)
printf("%s", buf)

reads until a random 0x00 byte



OOB READ

Reads past buf → leaks canary / ret addr / crash

A length check that passes ≠ a safe, valid string — validate the type & sign, and NUL-terminate explicitly.

Unsigned wraparound in the allocation size

the buffer shrinks, but the copy is still full-size

```
void func(size_t len, char *data) {
char *buf = malloc(len + 2);
if (buf == NULL) return;
memcpy(buf, data, len);
buf[len] = '\n';
buf[len + 1] = '\0';
}
```

① size can wrap

② full-size copy

len = SIZE_MAX
0xFFFF…FFFF → len + 2 wraps = 1 → malloc(1) 1-byte buffer → memcpy copies len ≈ 1.8×10¹⁹ B → HEAP OVERFLOW

Why len + 2 becomes 1

```
0xFFFFFFFFFFFFFFFF   SIZE_MAX
+ 2
------------------
1 0000000000000001
```

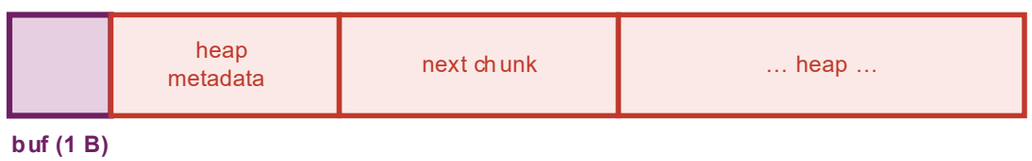
carry out of 64 bits discarded (mod 2^{64})

0x0000000000000001 = 1

a tiny number — malloc allocates almost nothing

1-byte buffer, full-size copy

memcpy writes ≈ 1.8×10¹⁹ bytes →



overruns the tiny chunk, corrupting heap metadata & neighbors → crash / RCE

Untrusted len in malloc(len + k): the sum can wrap → tiny allocation, full-size copy → overflow. Fix: if (len > SIZE_MAX - 2) return;