



hexhive

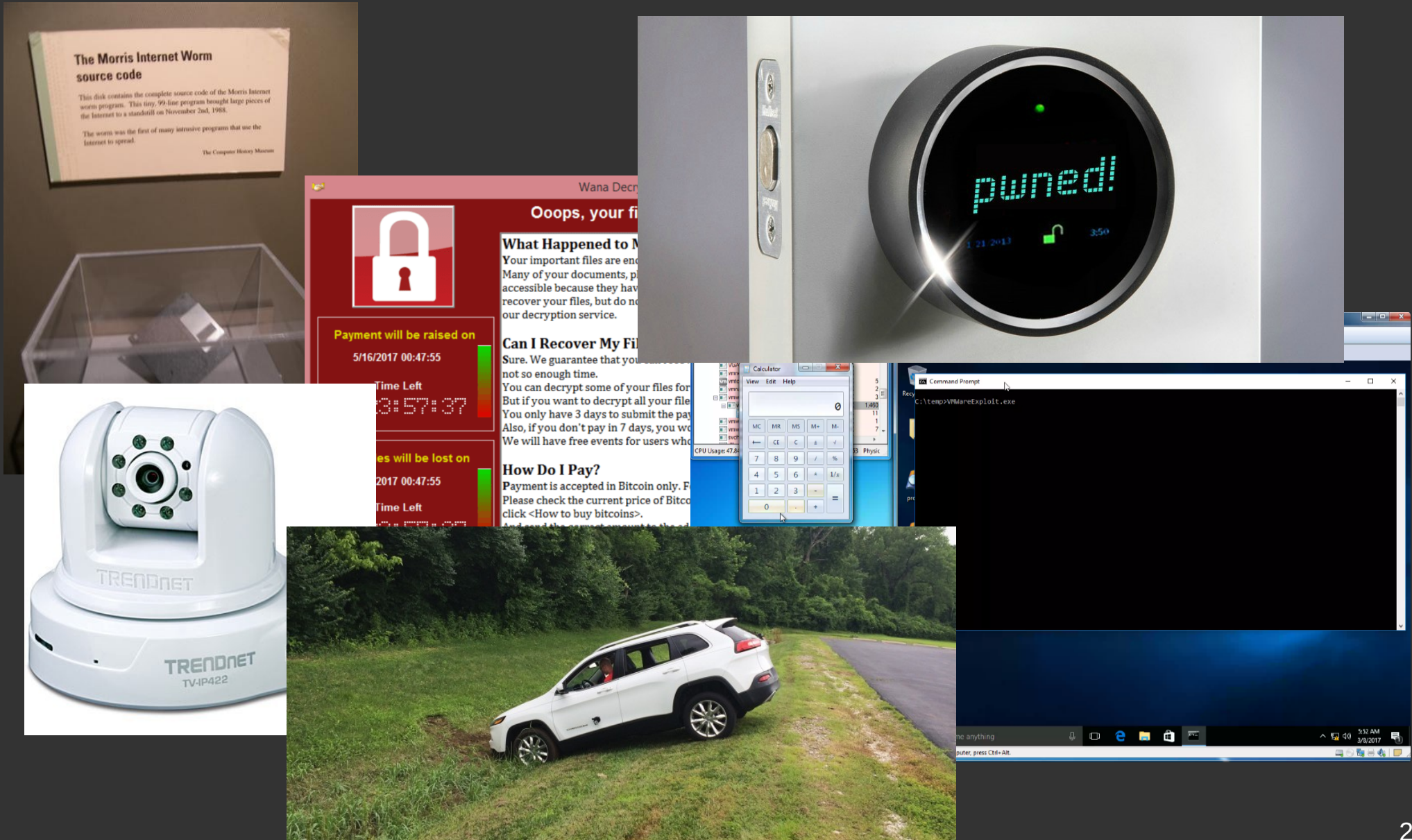
COM-301 Computer Security

Software Security

(AKA Defense)

Mathias Payer <mathias.payer@epfl.ch>
<http://hexhive.github.io>

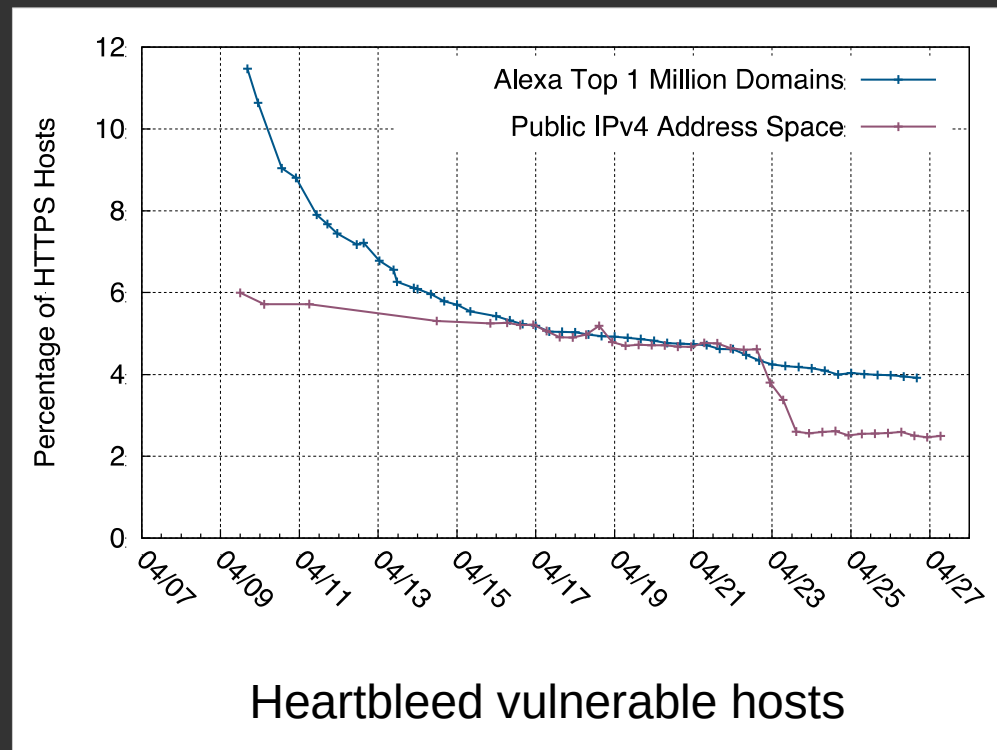
Vulnerabilities everywhere?



Heartbleed: reactive patching*



- 11% of servers remained vulnerable after 48 hours
- Patching plateaued at 4%
- Only 10% of vulnerable sites replaced certificates
- 15% of replaced cert's used vulnerable keys



* The Matter of Heartbleed. Zakir Durumeric, James Kasten, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Bernhard Amann, Jethro Beekman, Mathias Payer, Vern Paxson. In ACM IMC'14 (best paper)

Software is highly complex

Low-level languages (C/C++) trade type safety and memory safety for performance

Google Chrome: 76 MLoC

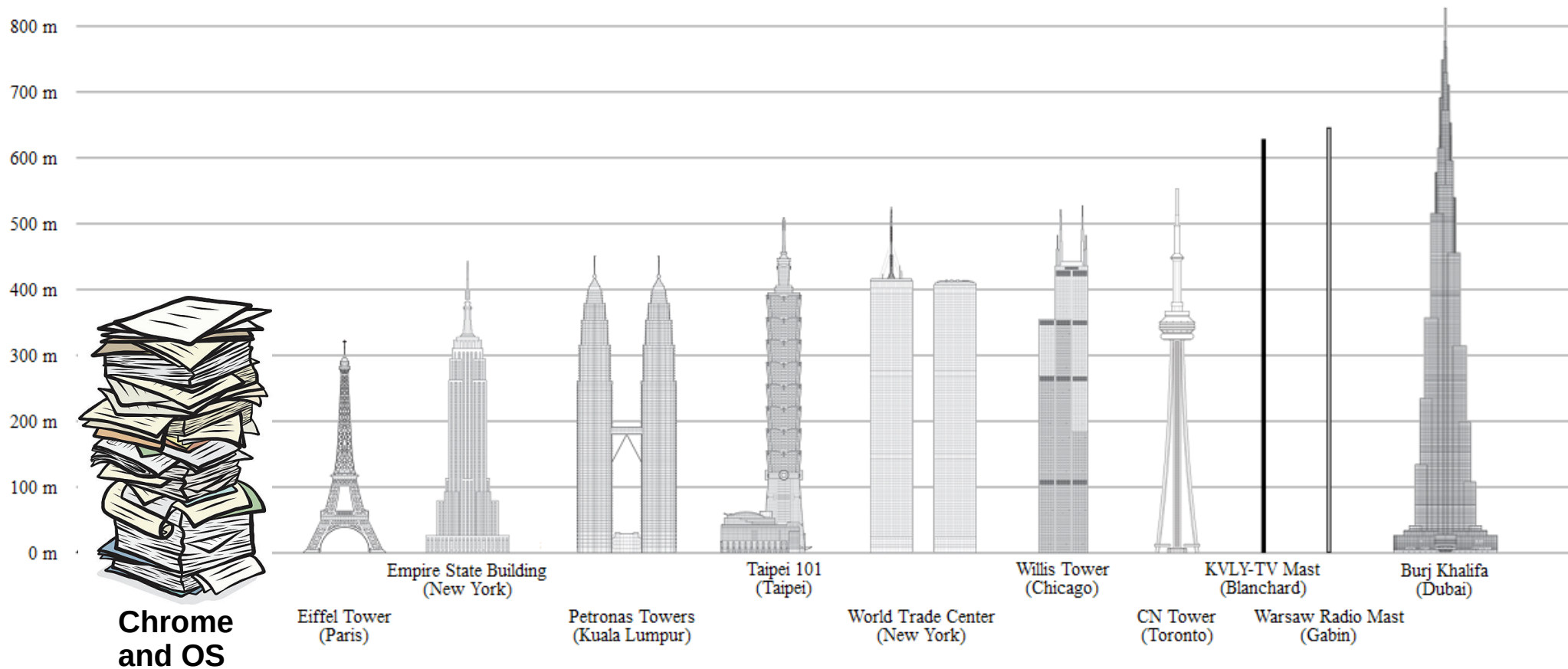
Gnome: 9 MLoC

Xorg: 1 MLoC

glibc: 2 MLoC

Linux kernel: 17 MLoC

Software is highly complex



~100 mLoC, 27 lines/page, 0.1mm/page \approx 370m

Defense: Mitigations vs. Testing



Mitigations

- Stop exploitation
- Always on
- Low overhead



Software Testing

- Discover bugs
- Development tool
- Result oriented

Outline

- Introduction
- Memory Safety
- Mitigation
 - Code execution attacks
 - Basic mitigations
- Software Testing
 - Fuzzing
 - Sanitization
- Summary


Memory Safety (or the lack thereof)

Memory corruption

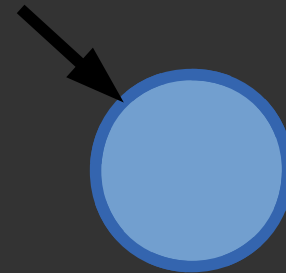

- Unintended modification of memory location due to missing / faulty safety check

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


Memory safety: temporal error



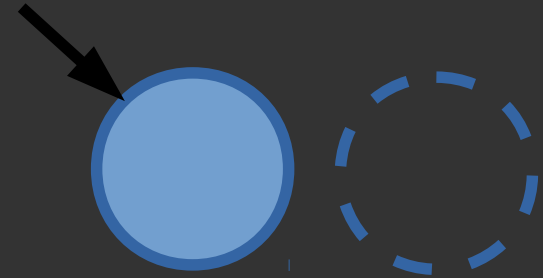
```
void vulnerable(char *buf) {  
    free(buf);  
    buf[12] = 42;  
}
```



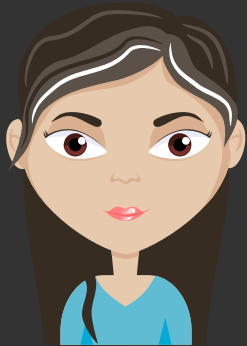
Memory safety: spatial error



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

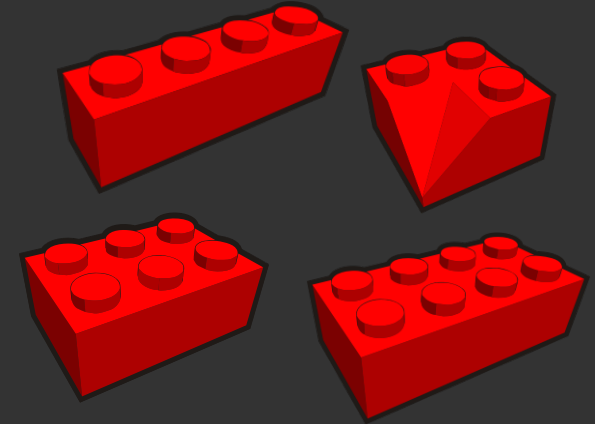



Problem: broken abstractions



C/C++

```
void log(int a) {  
    printf("Log: ");  
    printf("%d", a);  
}  
void (*fun)(int) = &log;  
void init() {  
    fun(15);  
}
```



ASM

```
log:  
    ...  
fun:  
    .quad log  
init:  
    ...  
    movl $15, %edi  
    movq fun(%rip), %rax  
    call *%rax
```



Mitigations: Stop exploitation



Outline

- Introduction
- Memory Safety
- Mitigation
 - Code execution attacks
 - Basic mitigations
- Software Testing
 - Fuzzing
 - Sanitization
- Summary

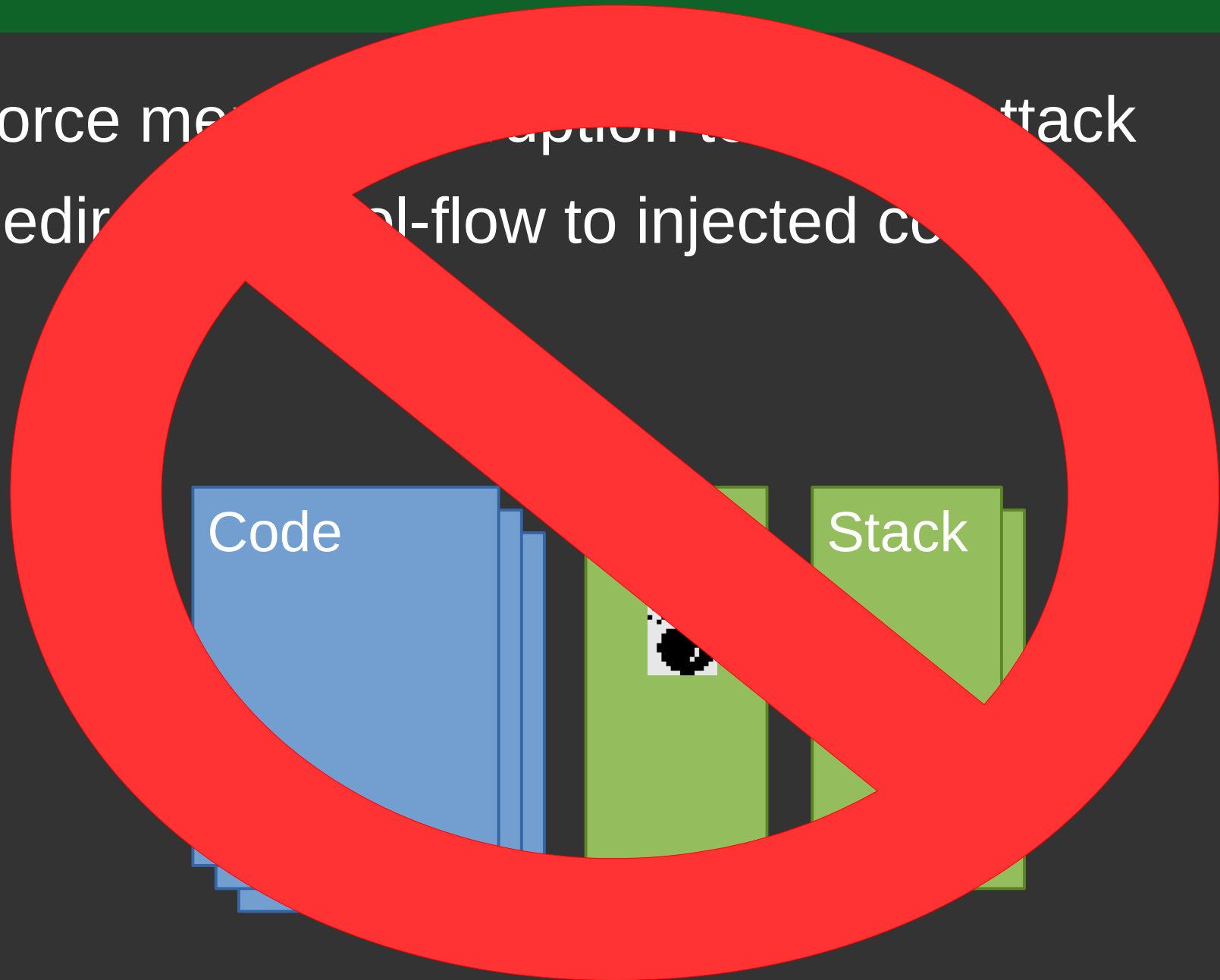
Modern Code Execution Attacks

Different threat models



Attack scenario: code injection

- Force memory execution to perform an 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);
```

Shellcode
(executable attack code)

Don't care

Points to shellcode

1st argument: *u1

Next stack frame

Memory safety Violation

Integrity

*C

Location

&C

Usage

*&C

Attack

Code
injection

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

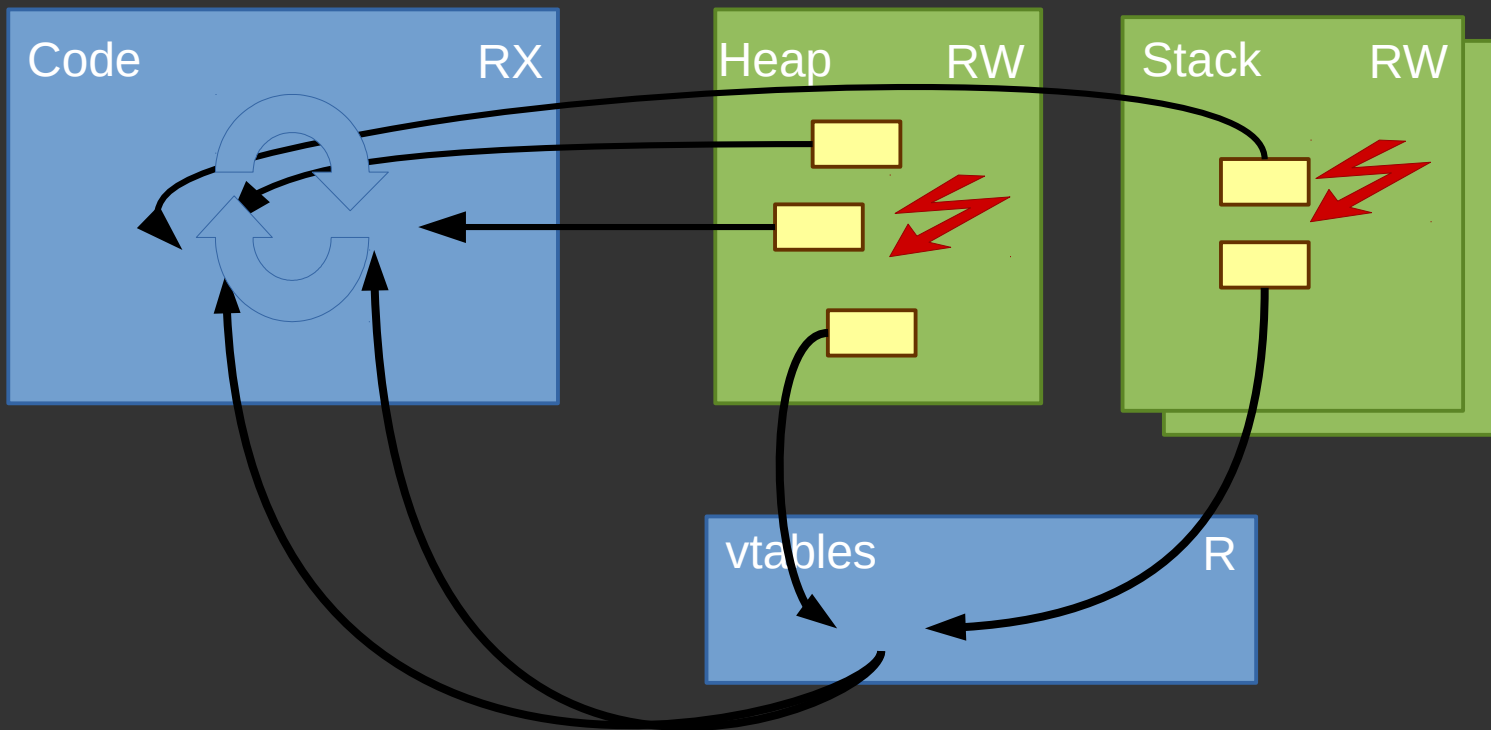
- Find addresses of gadgets
- Force memory corruption to set up attack
- Redirect control-flow to gadget chain



Attacker model: hijacking control-flow

Attacker may read/write arbitrary data

- Code is read-only, vtables are read-only
- Code pointers remain writable!



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()

1st argument to system()

Memory safety Violation

Integrity

*C

Location

&C

Usage

*&C

Attack

Control-flow
hijack

Address Space Layout Randomization

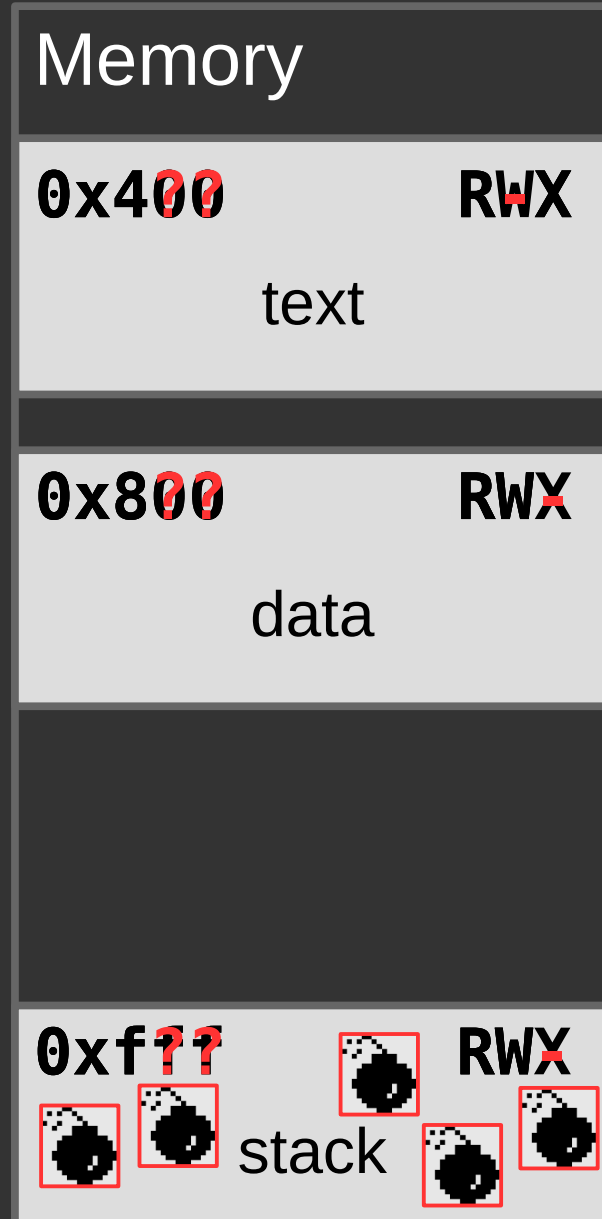
- Randomizes locations of code and data regions
 - Probabilistic defense
 - Depends on loader and OS
- Weaknesses and limitations
 - Prone to information leaks
 - Some regions remain static (on x86)
 - Performance impact (~10%)

Stack canaries

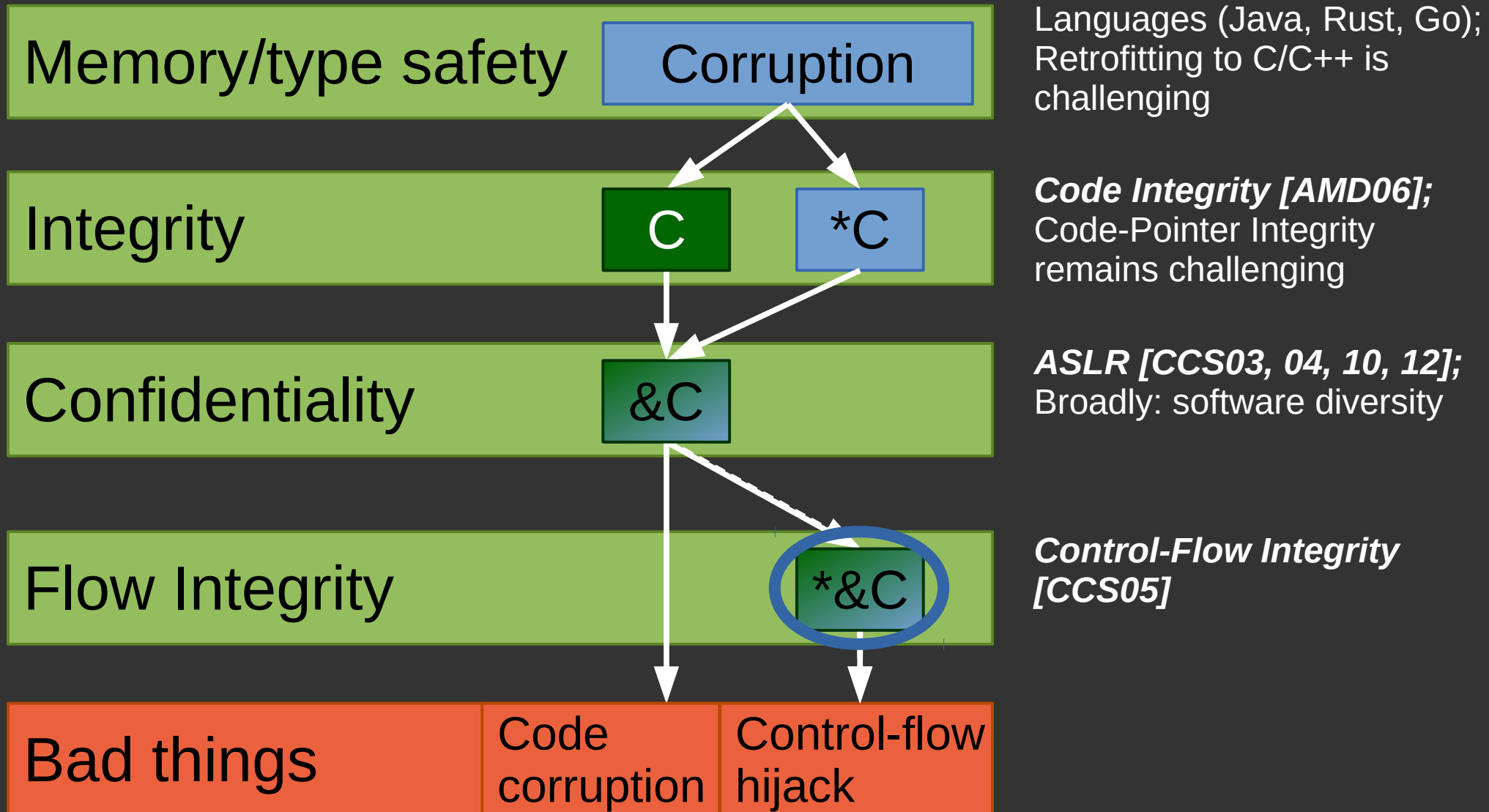
- Protect return instruction pointer on stack
 - Compiler modifies stack layout
 - Probabilistic protection
- Weaknesses and limitations
 - Prone to information leaks
 - No protection against targeted writes / reads

Status of deployed defenses

- Data Execution Prevention (DEP)
- Address Space Layout Randomization (ASLR)
- Stack canaries
- Safe exception handlers



Mitigation space in a nutshell [SP13]



Software Testing: Discover Bugs



Outline

- Introduction
- Memory Safety
- Mitigation
 - Code execution attacks and basic mitigations
 - Control-Flow Integrity
- Software Testing
 - Fuzzing
 - Sanitization
- Summary

Software Testing

- 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 a, b, c)
 - Operational requirements (performance, usability)
 - Security requirements?

Security testing?

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

- Complete testing of all control-flow/data-flow paths reduces to the halting problem
 - Control-flow: path through the program
 - Data-flow values used at each location
- Practical testing is limited by state explosion

Control-flow versus Data-flow

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

$a = 101$ and $a = 12$ covers all control-flow.

$a = 100$ results in a bug due to different data-

How to test security properties?

- Manual testing: search for bugs
 - Code review
 - Test cases
- Automatic testing
 - Develop analyses that discover bugs
 - Enforce security properties, test them

Forms of manual testing

- Exhaustive: cover all input
 - 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

Automatic bug finding

- 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 as a 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 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.

Today's standard is **branch coverage**

Is branch coverage complete?

Fuzzing



Fuzzing

- Fuzz testing (fuzzing) is an automated software testing technique. The fuzzing engine generates inputs based on some criteria:
 - Random mutation
 - Leveraging input structure
 - Leveraging program structure
- The inputs are then run on the test program and, if it crashes, a crash report is generated

Fuzzing: input generation

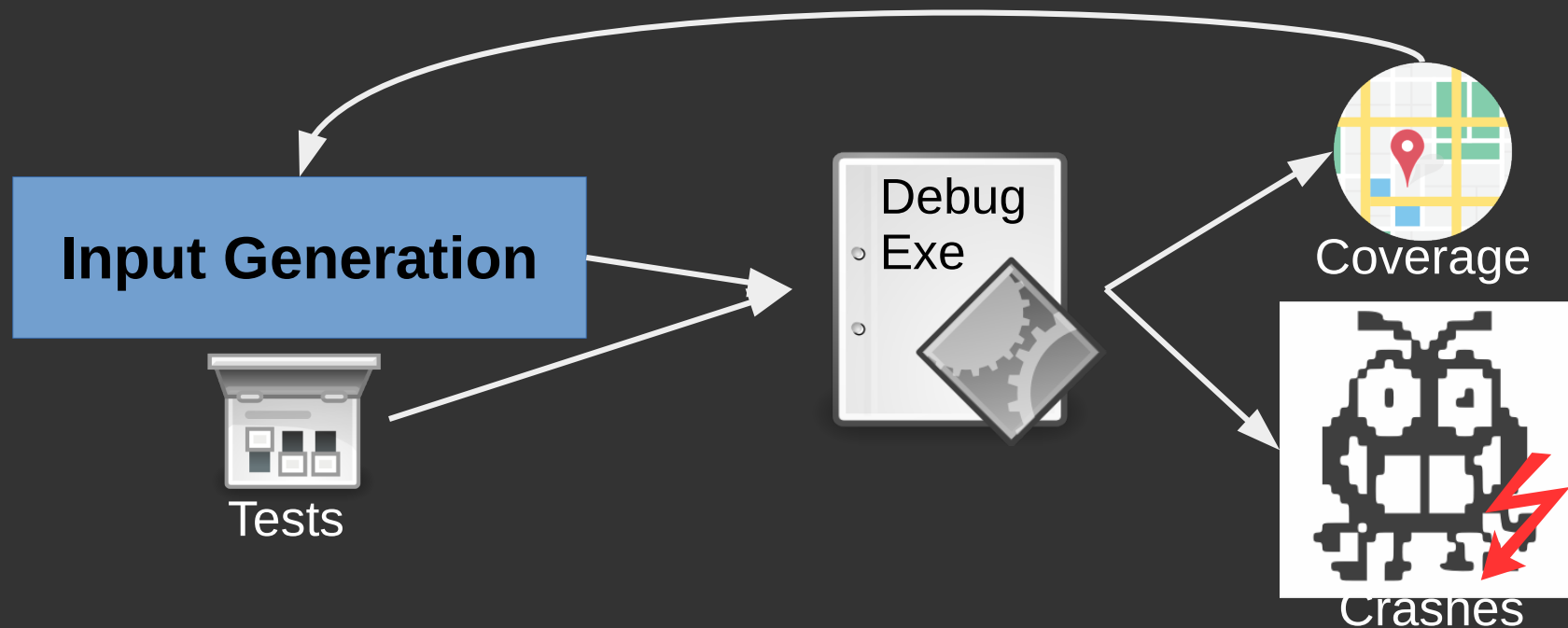
- ***Dumb Fuzzing*** is unaware of the input structure; randomly mutates input
- ***Generation-based fuzzing*** has a model that describes inputs; input generation produces new input seeds in each round
- ***Mutation-based fuzzing*** leverages a set of valid seed inputs; input generation modifies inputs based on feedback from previous rounds

Fuzzing: leverage program structure

- After execution, input can be modified based on program structure (and from past executions) to trigger new conditions
 - **White box** fuzzing leverages semantic program analysis to mutate input
 - **Grey box** leverages program instrumentation based on previous inputs
 - **Black box** fuzzing is unaware of the program structure

Fuzzing

- A random testing technique that mutates input to improve test coverage



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

FUZZ



ALL THE THINGS

memegenerator.net

Fuzzing as bug finding approach

- Fuzzing is highly effective bug finding (CVEs)
 - Proactive defense measure
 - First step in exploit development

OpenSSL



Different types of fuzzers

- Black box, generate random input
 - Set of valid samples will help! (e.g., Radamsa)
- Model-based: generate grammar-based input
 - Follows specification more closely
- Coverage-guided fuzzing, feedback loop
 - Push input generation to new coverage
 - AFL, Hongfuzz, libFuzzer

AFL: coverage-guided fuzzer

- Genetic algorithms to generate new input
- Simple, yet very effective: tons of security bugs
 - Take one input from queue
 - Minimize test case (as long as same behavior)
 - Mutate and execute
 - If new input: store sample in queue



Fuzzing: words of wisdom



Summary: Fuzzing

- Fuzzing is an effective way to automatically test programs for security violations (crashes)
 - Modern fuzzing uses coverage to guide input mutation, based on a generational strategy
 - Optimized for throughput
 - Fuzzers are easy to use and play with
- Most security bugs are found through fuzzing!

Sanitization



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)
- Typical 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.

Sanitizers

- AddressSanitizer:
<<https://clang.llvm.org/docs/AddressSanitizer.html>>
- LeakSanitizer: <<https://clang.llvm.org/docs/LeakSanitizer.html>>
- MemorySanitizer:
<<https://clang.llvm.org/docs/MemorySanitizer.html>>
- UndefinedBehaviorSanitizer:
<<https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>>
- ThreadSanitizer:
<<https://clang.llvm.org/docs/ThreadSanitizer.html>>
- HexType: <<https://github.com/HexHive/HexType>>
- Use sanitizers to test your code. More sanitizers are in development.

Summary: Sanitizers

- Sanitizers enforce a runtime policy
 - Make undefined/illegal behavior explicit
 - Detect violations when they happen, not later
- Efficient sanitization policies
 - Memory safety
 - Undefined behavior
- Sanitizers can be combined with fuzzing to detect more crashes

Outline

- Introduction
- Memory Safety
- Mitigation
 - Code execution attacks and basic mitigations
 - Control-Flow Integrity
- Software Testing
 - Fuzzing
 - Sanitization
- Summary

Are we making progress?



2007



2017

Summary: software security

- Two approaches: mitigation and testing
- Mitigations stop unknown vulnerabilities
 - Make exploitation harder, not impossible
 - Focus on control-flow hijacking
 - CFI: stateless, depends on precision of analysis
- Testing discovers bugs during development
 - Automatically generate test cases through fuzzing
 - Make bug detection more likely through sanitization

Thank you! Questions?

