# Software Security Techniques

CMSC 23200/33250, Winter 2022, Lecture 6


David Cash and Blase Ur

University of Chicago

# Software security, so far this quarter

Buggy programs are common, so hardware, OS and compiler are designed to contain damage.

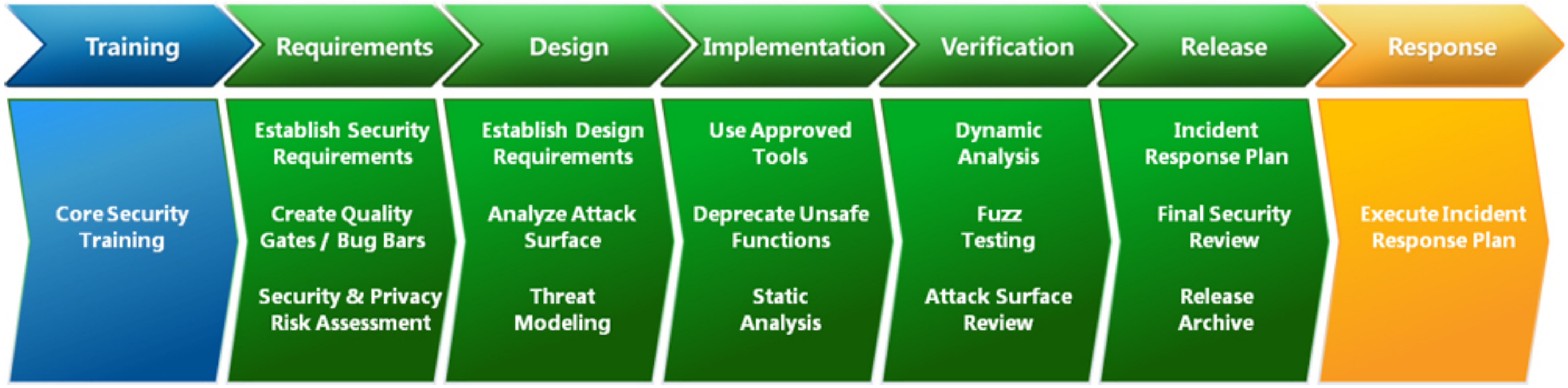
1. Hardware protection: Privileged mode
2. Process isolation via virtual memory
3. Stack Protectors
4. Address-space layout randomization
5. Write-XOR-Execute

This lecture: Preventing or catching bugs earlier.

# Secure Software Development

Microsoft "Secure Software Development Lifecycle" (2004-present)

- Training
- Design security requirements
- Metrics & compliance reporting
- Threat modeling
- Establish design requirements
- Define & use crypto standards
- Manage risk of third-party components
- Use approved tools
- Static analysis security testing
- Dynamic analysis security testing
- Penetration testing
- Incident response

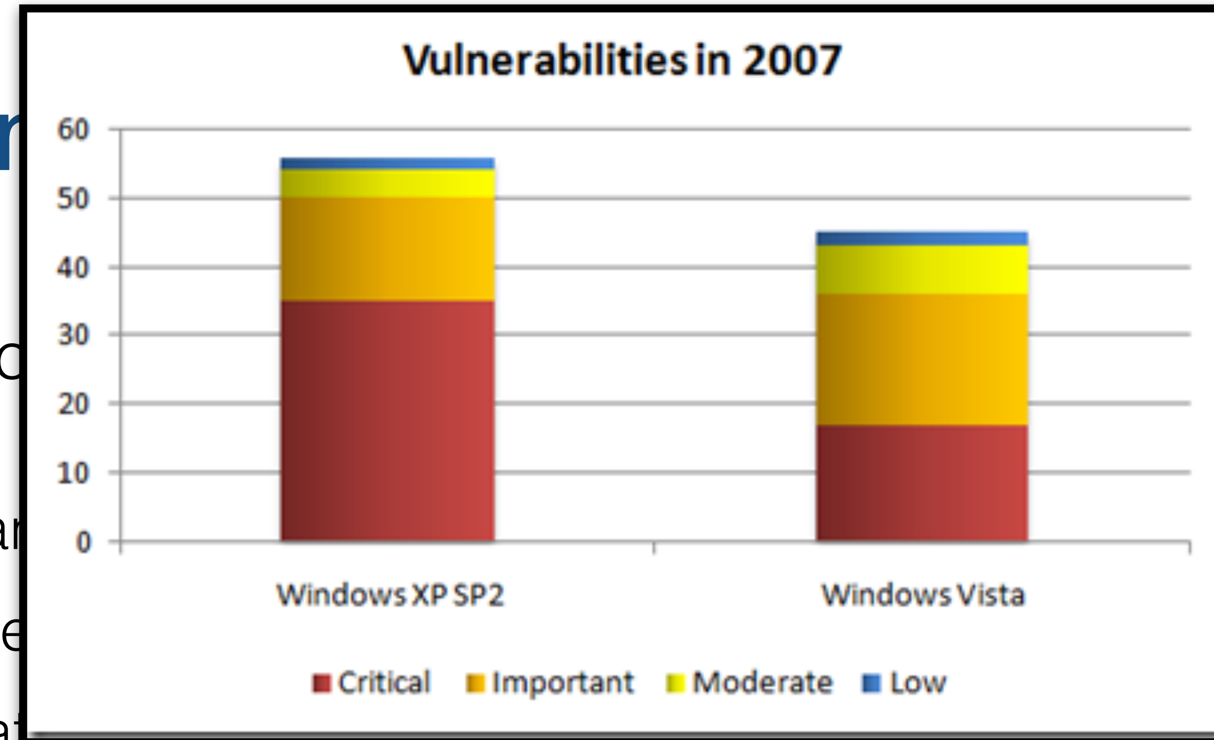


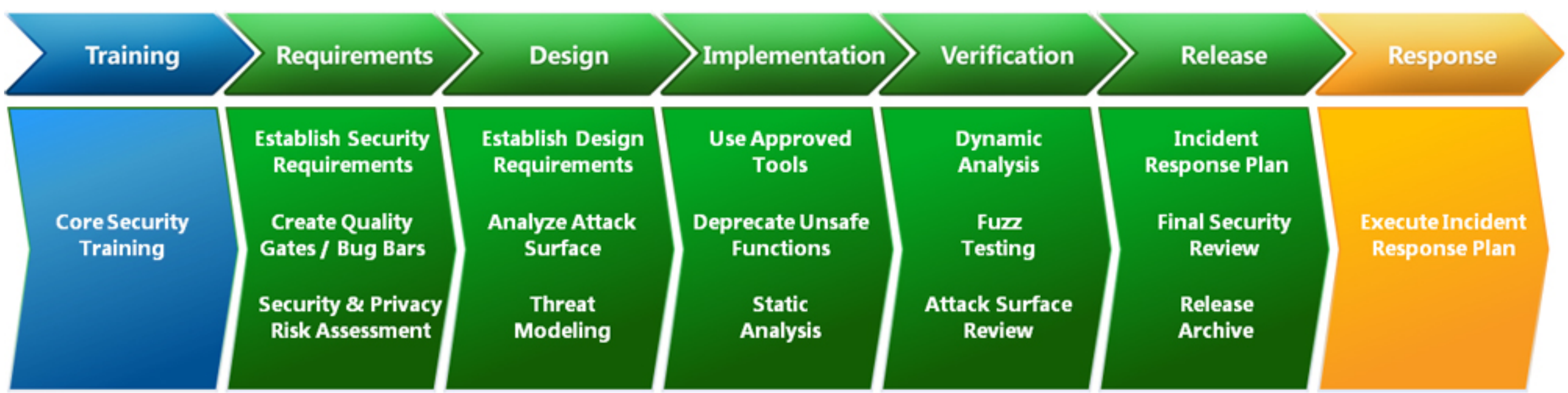
# Secure Software Developmen

Microsoft “Secure Software Development Lifec

- Training
- Design security requirements
- Metrics & compliance reporting
- Threat modeling
- Establish design requirements
- Define & use crypto standards

- Mar
- Use
- Stat
- Dynamic analysis security testing
- Penetration testing
- Incident response



# Memory-Safe Languages

Many of our problems can be solved by using "memory-safe" languages.

> *Memory safety is the state of being protected from various software bugs and security vulnerabilities when dealing with memory access, such as buffer overflows and dangling pointers.*
>
> *-Wikipedia*

The model of execution for such languages simply *does not allow* for such bugs.

| Not Memory-Safe | Memory Safe |
|---|---|
| C | Java |
| C++ | Python |
| Assembly | Javascript |
| | Rust, Go, Haskell, … |

Should be avoided if at all possible, but lots of legacy code (and low-level stuff).

# Software is Complex

All written in unsafe C/C++/Assembly

| Project | Lines of Code | No. Contributors |
|---|---|---|
| Apache HTTP Server | 1.5 million | 125 |
| Apache OpenOffice | 9 million | 140 |
| Linux Kernel | 19 million | 14,000 |
| OpenSSL | 600k | 572 |

# Example Bug: Heartbleed in OpenSSL

OpenSSL is a very widely-used library for TLS, the main security protocol on the internet. Used in Apache, Nginx, …

**March 2014**: Researchers discover vulnerability in "heartbeat" implementation.

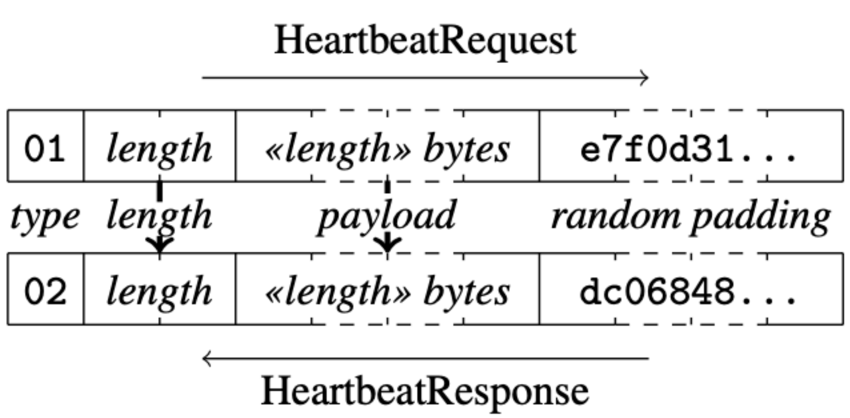
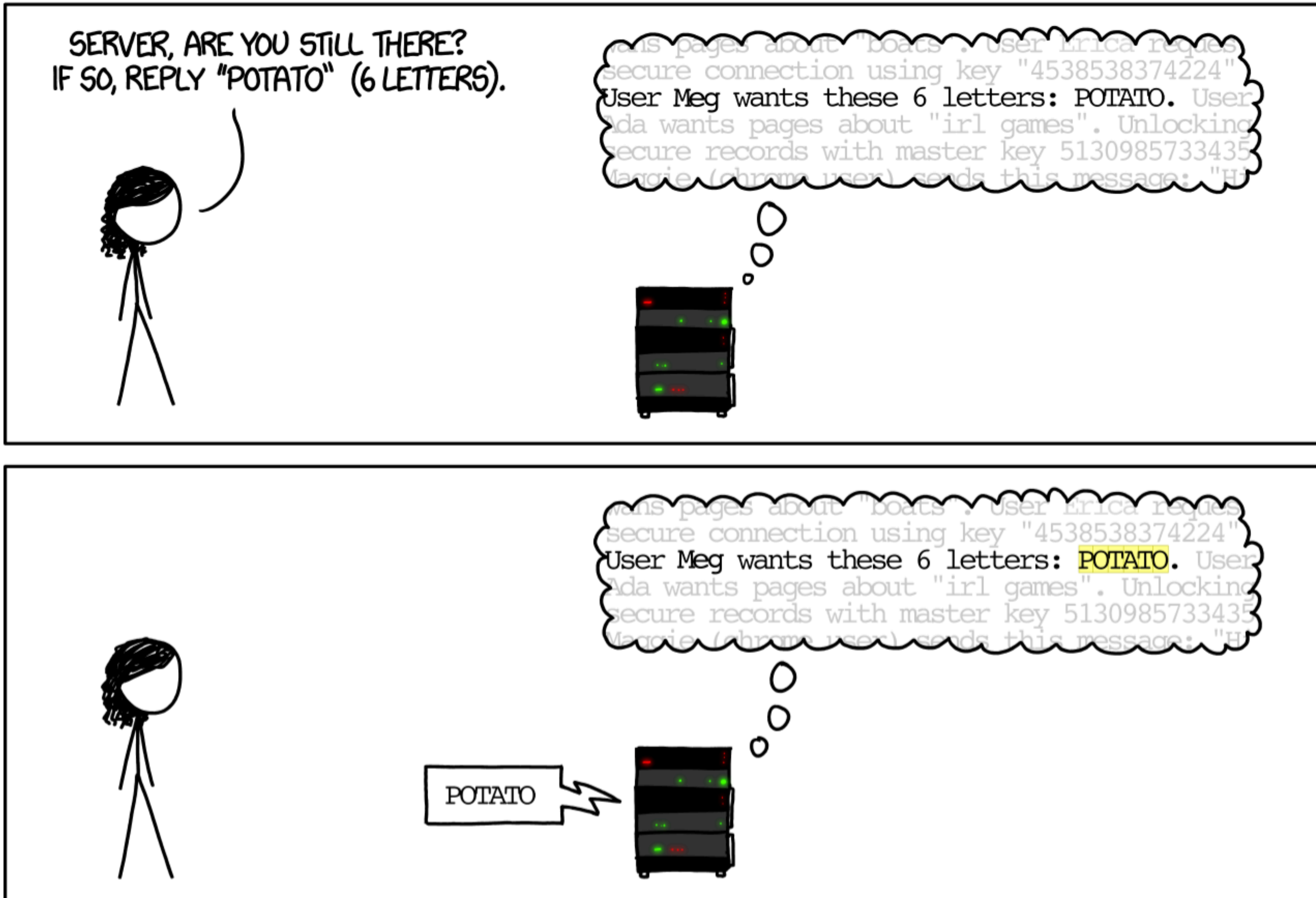
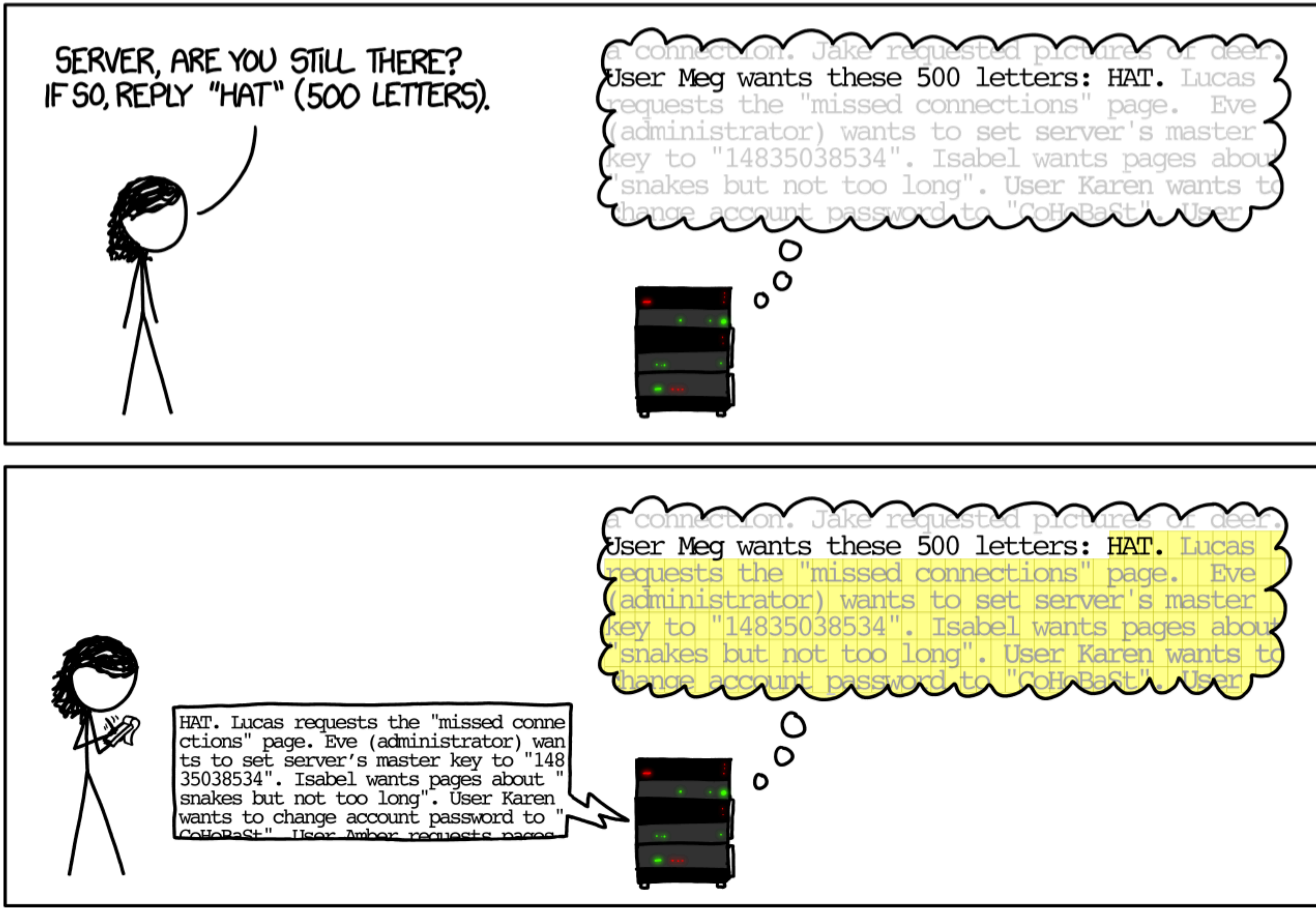



Figure 1: **Heartbeat Protocol.** Heartbeat requests include user data and random padding. The receiving peer responds by echoing back the data in the initial request along with its own padding.

*Credit: [Durumeric et al 2014]*


SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
secure connection using key "4538538374224"
User Meg wants these 6 letters: POTATO. User
Ada wants pages about "irl games". Unlocking
secure records with master key 5130985733435
secure connection using key "4538538374224"
User Meg wants these 6 letters: POTATO. User
Ada wants pages about "irl games". Unlocking
secure records with master key 5130985733435
POTATO

Credit: xkcd


SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "

*Credit: xkcd*

# Heartbleed

```c
int
dtls1_process_heartbeat(SSL *s)
{
    unsigned char *p = &s->s3->rrec.data[0];
    unsigned char *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */

    //...

    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;

    // ...

    unsigned char* buffer;
    unsigned char* bp;
    int r;

    /*
     * Allocate memory for the response, size is 1 byte message type, plus 2
     * byte payload length, plus payload, plus padding
     */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;

    // ...

    /* Enter response type, length and copy payload */
    *bp++ = TLS_HB_RESPONSE;
    s2n(playload, bp);
    memcpy(bp, pl, payload);

}
```

Uses un-sanitized length as input to memcpy.

# Discovery of Heartbleed

"I was doing laborious auditing of OpenSSL, going through the [Secure Sockets Layer] stack line by line.

| Date | Event |
|---|---|
| 03/21 | Neel Mehta of Google discovers Heartbleed |
| 03/21 | Google patches OpenSSL on their servers |
| 03/31 | CloudFlare is privately notified and patches |
| 04/01 | Google notifies the OpenSSL core team |
| 04/02 | Codenomicon independently discovers Heartbleed |
| 04/03 | Codenomicon informs NCSC-FI |
| 04/04 | Akamai is privately notified and patches |
| 04/05 | Codenomicon purchases the `heartbleed.com` domain |
| 04/06 | OpenSSL notifies several Linux distributions |
| 04/07 | NCSC-FI notifies OpenSSL core team |
| 04/07 | OpenSSL releases version 1.0.1g and a security advisory |
| 04/07 | CloudFlare and Codenomicon disclose on Twitter |
| 04/08 | Al-Bassam scans the Alexa Top 10,000 |
| 04/09 | University of Michigan begins scanning |

*Credit: [Durumeric et al 2014]*

# Many Systems were Vulnerable to Heartbleed

Scripts automatically tested Alex top 1-million sites

| Web Server | Alexa Sites | Heartbeat Ext. | Vulnerable |
|---|---|---|---|
| Apache | 451,270 (47.3%) | 95,217 (58.4%) | 28,548 (64.4%) |
| Nginx | 182,379 (19.1%) | 46,450 (28.5%) | 11,185 (25.2%) |
| Microsoft IIS | 96,259 (10.1%) | 637 (0.4%) | 195 (0.4%) |
| Litespeed | 17,597 (1.8%) | 6,838 (4.2%) | 1,601 (3.6%) |
| Other | 76,817 (8.1%) | 5,383 (3.3%) | 962 (2.2%) |
| Unknown | 129,006 (13.5%) | 8,545 (5.2%) | 1,833 (4.1% ) |

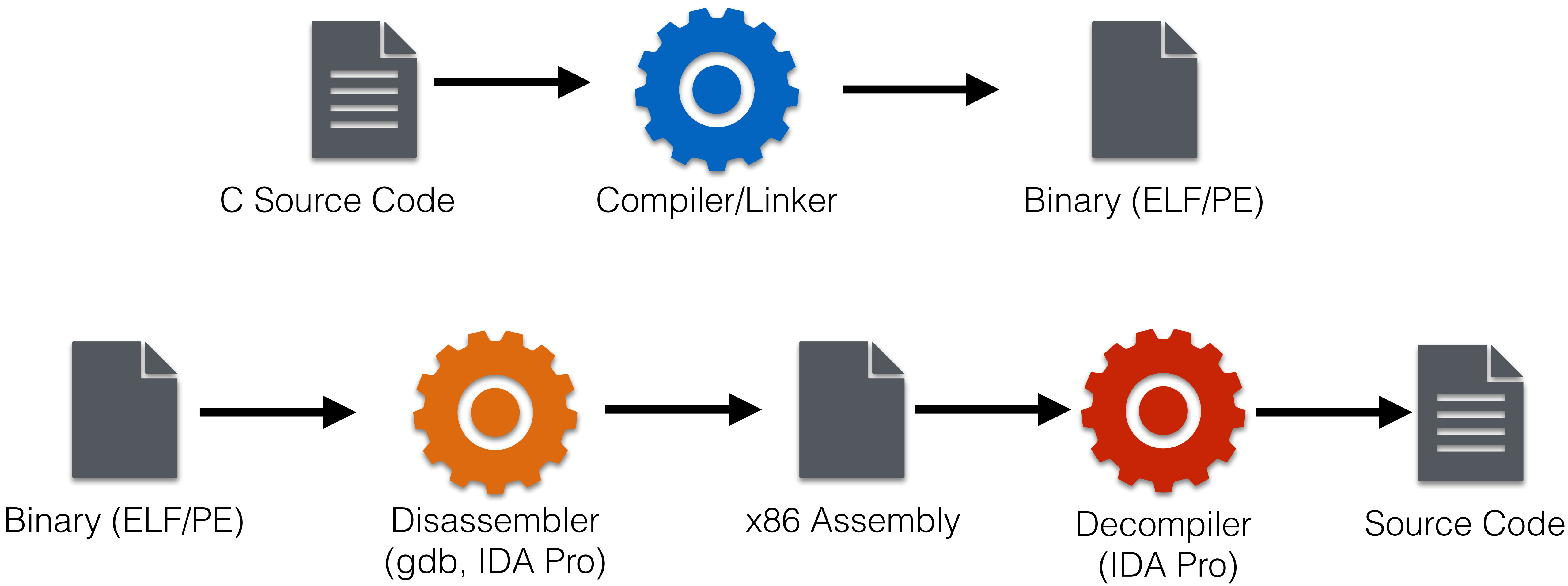
*Credit: [Durumeric et al 2014]*

# Many Systems were Vulnerable to Heartbleed

| Site | Vuln. | Site | Vuln. | Site | Vuln. |
|---|---|---|---|---|---|
| Google | Yes | Bing | No | Wordpress | Yes |
| Facebook | No | Pinterest | Yes | Huff. Post | ? |
| Youtube | Yes | Blogspot | Yes | ESPN | ? |
| Yahoo | Yes | Go.com | ? | Reddit | Yes |
| Amazon | No | Live | No | Netflix | Yes |
| Wikipedia | Yes | CNN | ? | MSN.com | No |
| LinkedIn | No | Instagram | Yes | Weather.com | ? |
| eBay | No | Paypal | No | IMDB | No |
| Twitter | No | Tumblr | Yes | Apple | No |
| Craigslist | ? | Imgur | Yes | Yelp | ? |

*Credit: [Durumeric et al 2014]*

# Finding Bugs in a Binary is Even Harder




Relatively simple programs like `strings` and `hexdump` can be a start.

But binary analysis is often used for reverse engineering and malware analysis.

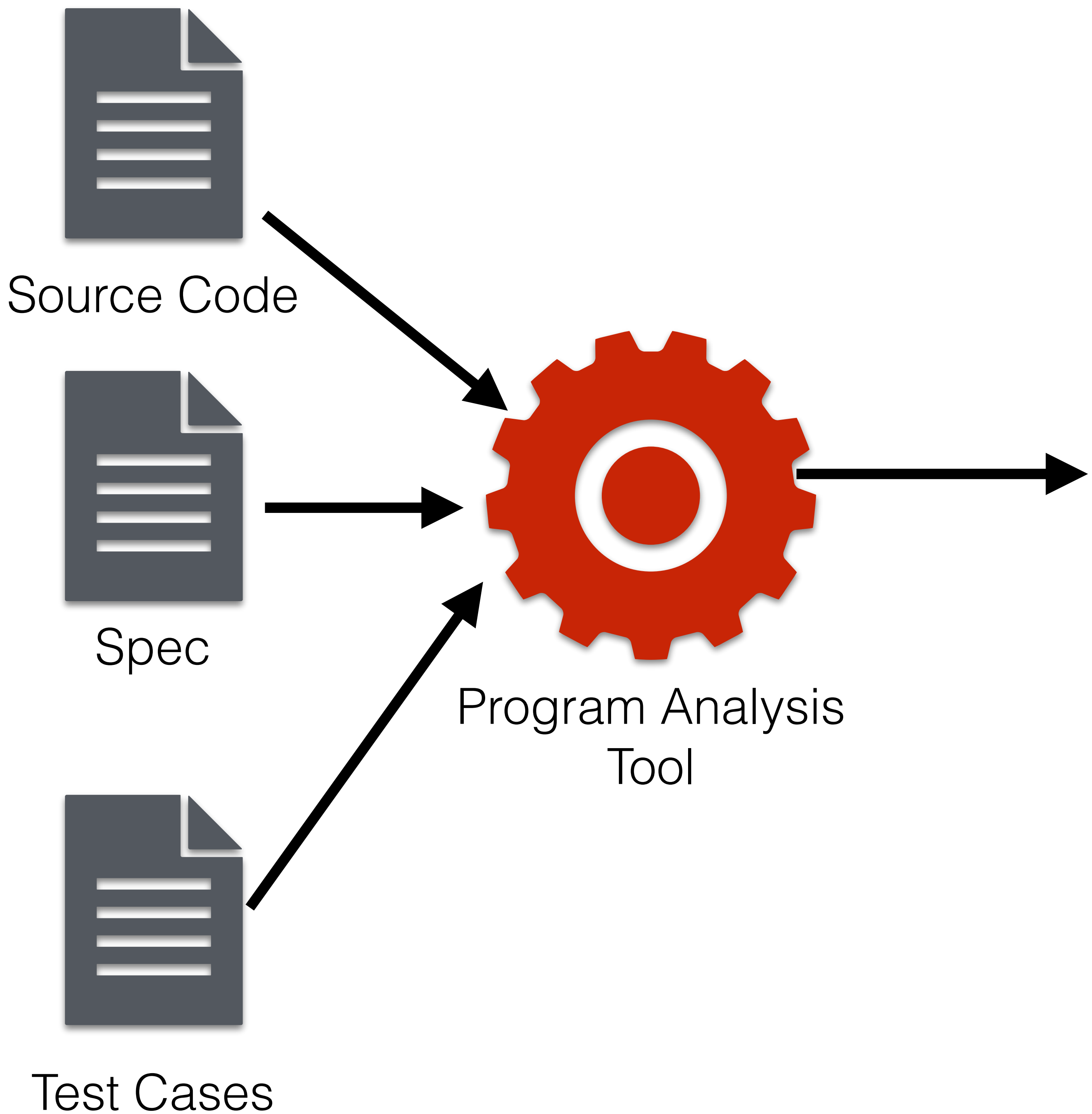
# Disassembly/Decompiling Can Find Bugs

```
Dump of assembler code for function main:
   0x0804843b <+0>:  lea    0x4(%esp),%ecx
   0x0804843f <+4>:  and    $0xfffffff0,%esp
   0x08048442 <+7>:  pushl  -0x4(%ecx)
   0x08048445 <+10>: push   %ebp
   0x08048446 <+11>: mov    %esp,%ebp
   0x08048448 <+13>: push   %ecx
=> 0x08048449 <+14>: sub    $0x14,%esp
   0x0804844c <+17>: sub    $0xc,%esp
   0x0804844f <+20>: push   $0x40
   0x08048451 <+22>: call   0x8048310 <malloc@plt>
   0x08048456 <+27>: add    $0x10,%esp
   0x08048459 <+30>: mov    %eax,-0xc(%ebp)
   0x0804845c <+33>: sub    $0xc,%esp
   0x0804845f <+36>: pushl  -0xc(%ebp)
   0x08048462 <+39>: call   0x8048300 <free@plt>
   0x08048467 <+44>: add    $0x10,%esp
   0x0804846a <+47>: sub    $0xc,%esp
   0x0804846d <+50>: pushl  -0xc(%ebp)
   0x08048470 <+53>: call   0x8048300 <free@plt>
   0x08048475 <+58>: add    $0x10,%esp
   0x08048478 <+61>: mov    $0x0,%eax
   0x0804847d <+66>: mov    -0x4(%ebp),%ecx
   0x08048480 <+69>: leave
   0x08048481 <+70>: lea    -0x4(%ecx),%esp
   0x08048484 <+73>: ret
End of assembler dump.
```

# Techniques for Bug Finding with Source

1. Manual Analysis
   - Source review
   - Reverse engineering
2. Automated Program Analysis
   - Static Analysis
   - Dynamic Analysis (Testing)

# Source Code Analyzers




| Report No. | Type | Line/test |
|---|---|---|
| 1 | Memory leak | 125 |
| 2 | Buffer overflow | 386 |
| 3 | Use-after-free | 776 |
| 4 | Info leak | 432 |
| 5 | Unsanitized input | 321 |
| … | … | … |

# False Positives and Negatives in Program Analysis

| Term | Definition |
|---|---|
| False positive | Spurious warning when there is no vulnerability |
| False negative | Lack of warning when for actual vulnerability |

| Term | Definition |
|---|---|
| Complete Analysis | No false negatives |
| Sound Analysis | No false positives |

# Complete and Sound Analysis?

<u>Rice's Theorem from Computability Theory (informal):</u> Any non-trivial behavioral property of a programs' behavior is *undecidable*.

<u>Examples:</u> Given a program P, will it…

- Always give correct output?
- Go into an infinite loop?
- Segfault?
- Leak memory?
- …

*(<u>Technical disclaimers</u>: Rice's theorem only applies to "programs" with unbounded memory, and not to the ones in our computers, strictly speaking. Nonetheless the conclusion is still true in practice.)*

May be possible to check those properties for simple programs, however!

# Typical Tools/Approaches

| Approach | Type | Comment |
|---|---|---|
| Lexical analyzer | Static | Perform syntactic checks<br><br>Ex: grep, LINT, RATS, ITS4 |
| Fuzzing | Dynamic | Run program on many possibly-malformed inputs.<br><br>Ex: AFL/libfuzzer, Grizzly, Taof, |
| Run-time instrumentation | Dynamic | Add correctness checks to binary by simulating in VM, replacing standard libraries.<br><br>Ex: Valgrind/Memcheck |
| Compile-time instrumentation | Static/Dynamic | Insert checks into binary during compilation.<br><br>Ex: {Address,Thread,UndefinedBehavior}Sanitizer |
| Symbolic Execution | Static/Dynamic | Abstract behavior of program then algebraically solve for buggy inputs.<br><br>Ex: KLEE, S2E, FiE |
| Model Checking | Static | Define a specification, abstract program to model, then formally verify correctness.<br><br>Ex: MOPS, SLAM, … |

# Lexical Analysis: Source Code Scanners

- `grep` (i.e. simply search) for "strcpy" to find use of unsafe code.
- `lint` searches for problematic code features
- `RATS/ITS4:` more modern versions of this approach.

  - Some array out-of-bounds errors
  - Ignoring return values
  - Variables that can be static but aren't
  - Unsanitized integer/string inputs
  - Missing optional args (e.g. in `open()`)
  - …

# Compile-Time Instrumentation: AddressSanitizer (ASan)

`-fsanitize=address` option in `gcc` will insert numerous checks to binary

Ex: Rewrite mallocs to ask for extra memory, then mark bytes before/after as “redzone”. Touching those indicates error.

Before:

```
*address = ...;  // or: ... = *address;
```

After:

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;  // or: ... = *address;
```

```
void foo() {
  char a[8];
  ...
  return;
}
```

Instrumented code:

```
void foo() {
  char redzone1[32];  // 32-byte aligned
  char a[8];          // 32-byte aligned
  char redzone2[24];
  char redzone3[32];  // 32-byte aligned
  int  *shadow_base = MemToShadow(redzone1);
  shadow_base[0] = 0xffffffff;  // poison redzone1
  shadow_base[1] = 0xffffff00;  // poison redzone2, unpoison 'a'
  shadow_base[2] = 0xffffffff;  // poison redzone3
  ...
  shadow_base[0] = shadow_base[1] = shadow_base[2] = 0; // unpoison all
  return;
}
```

Source: https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm

(Demo)

# Dynamic Analysis: Valgrind

**`Valgrind/Memcheck`** will rewrite a *binary* with many checks for memory errors.

Does not catch as much as ASAN, general. For example stack bugs get through:

```
// RUN: clang -O -g -fsanitize=address %t && ./a.out
int main(int argc, char **argv) {
  int stack_array[100];
  stack_array[1] = 0;
  return stack_array[argc + 100];  // out of bounds
}
```

Source: https://github.com/google/sanitizers/wiki/AddressSanitizerExampleStackOutOfBounds

(Demo)

# Program Fuzzing

Run program on huge number of automatically-generated inputs, searching for crashes.

Linux Mint fixes screensaver bypass discovered by two kids

Two children playing on their dad's computer accidentally found a way to bypass the screensaver and access locked systems.

By Catalin Cimpanu for Zero Day | January 15, 2021 -- 18:28 GMT (10:28 PST) | Topic: Security

MORE FROM CATALIN CIMPANU

Security
Hacker leaks data of

"A few weeks ago, my kids wanted to hack my Linux desktop, so they typed and clicked everywhere while I was standing behind them looking at them play," wrote a user identifying themselves as robo2bobo.

According to the bug report, the two kids pressed random keys on both the physical and on-screen keyboards, which eventually led to a crash of the Linux Mint screensaver, allowing the two access to the desktop.

"I thought it was a unique incident, but they managed to do it a second time," the user added.

# Types of Fuzzing

**Mutation-based (dumb)**: Take an initial set of examples and make random changes to them.

- Millions of inputs (can just run forever)
- Possibly lower quality, unable to find certain types of inputs

**Generative (smart)**: Describe inputs to fit format/protocol, then generate inputs from that grammar with changes.

- Run with fewer inputs, which can be directed to certain types

**Q:** Which is better for `func()`?

**Q:** Which is better for heart bleed?

```
int func(char *s) {
  if(check_sum_is_valid(s)) {
    complicated_func(s);
   }
  else {
    simple_func(s);
   }
}
```

# Problems with Fuzzing

**Mutation-based (dumb)**: How long to run? And we need a strong server.

**Generative (smart)**: Run out test cases. A lot more work.

**General problems**:

— Need to identify when bug/crash occurs automatically.

— Don't want to report same bug 1000s of times.

# Fuzzing and Code Coverage

**Testing heuristic**: The more of the code that is executed by tests, the more likely we are to find bugs.

Can try to cover:

- Lines/instructions of source/binary
- Branches in binary/source
- Paths in binary/source

**Example:**

```
int func(int a, int b) {
  if(a > 2)
    a = 2;
  if(b > 2)
    b = 2;
  return a+b;
}
```

# A Notable Example: Dumb Mutation Fuzzing of PDFs

Charlie Miller, 2010:

1. Download 1000s of PDFs from internet

2. For each one, change some bytes literally at random.

```
numwrites = random.randrange(math.ceil((float(len(buf)) / FuzzFactor))) + 1
for j in range(numwrites):
 rbyte = random.randrange(256)
 rn = random.randrange(len(buf))
 buf[rn] = “%c"%(rbyte)
```

Results:

Apple Preview: 250 unique crashes, 60 exploits

Acrobat: 100 unique crashes, 4 exploits

*Slide credit: https://cs155.stanford.edu/lectures/06-testing.pdf*

# American Fuzzy Loop (AFL)

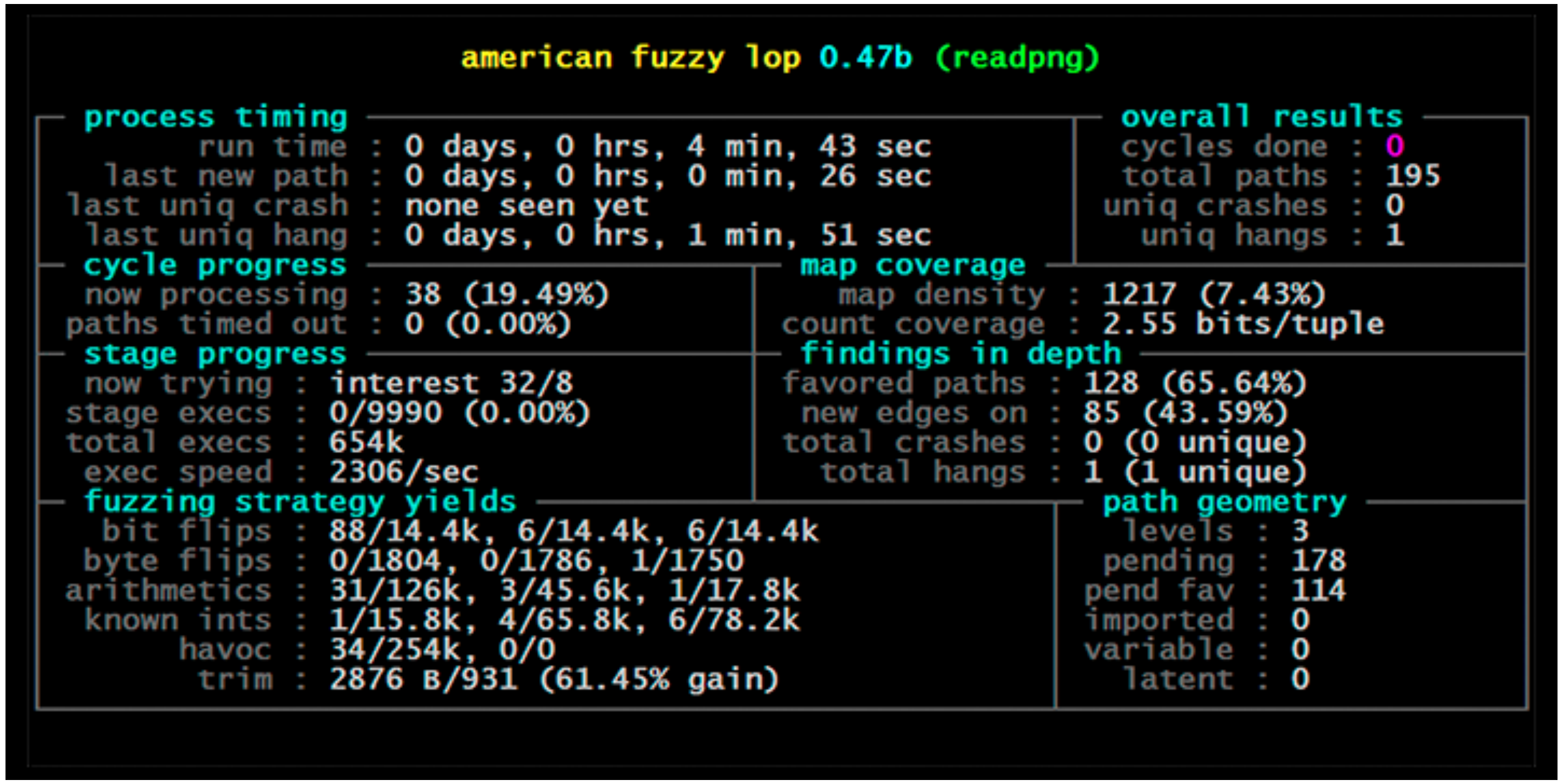
Popular, impactful project by Google.

Easy to set up with seed examples for mutation-based fuzzing.

Can instrument code for fast execution.

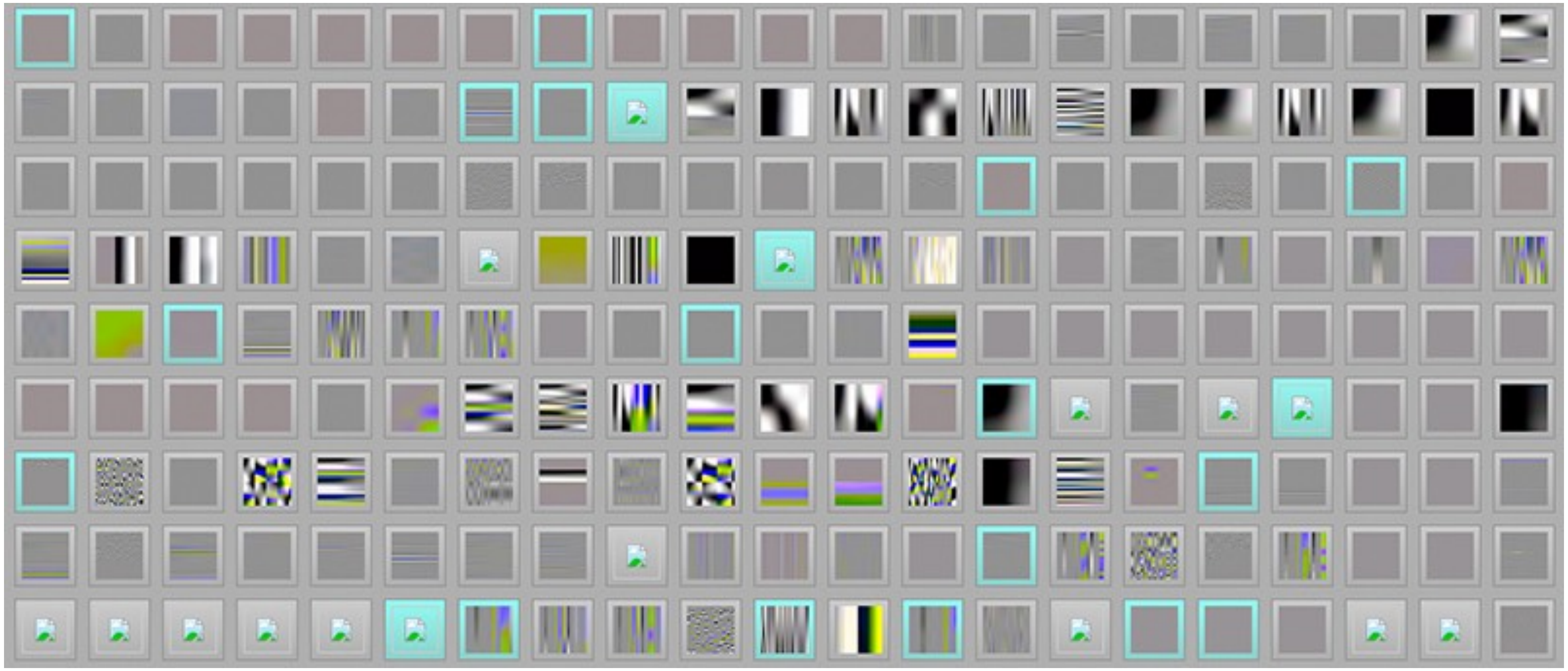
Deterministic bit-flipping, randomized stacked transforms.

Measures path coverage and favors increasing coverage.



# AFL Fuzz and File Formats

```
$ mkdir in_dir
$ echo 'hello' >in_dir/hello
$ ./afl-fuzz -i in_dir -o out_dir ./jpeg-9a/djpeg
```



Automatically discovered well-formed jpeg format by exploring code!

https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html

# Fuzzing in Production

Google/Microsoft constantly fuzz products with dedicated servers/VMS.

Anecdote: Found 95 vulnerabilities in Chrome during 2011.

## OneFuzz

### A self-hosted Fuzzing-As-A-Service platform

Project OneFuzz enables continuous developer-driven fuzzing to proactively harden software prior to release. With a single command, which can be baked into CICD, developers can launch fuzz jobs from a few virtual machines to thousands of cores.

## The bug-o-rama trophy case

Yeah, it finds bugs. I am focusing chiefly on development and have not been running the fuzzer at a scale, but here are some of the notable vulnerabilities and other uniquely interesting bugs that are attributable to AFL (in large part thanks to the work done by other users):

| | | |
|---|---|---|
| IJG jpeg 1 | libjpeg-turbo 1 2 | libpng 1 |
| libtiff 1 2 3 4 5 | mozjpeg 1 | PHP 1 2 3 4 5 6 7 8 |
| Mozilla Firefox 1 2 3 4 | Internet Explorer 1 2 3 4 | Apple Safari 1 |
| Adobe Flash / PCRE 1 2 3 4 5 6 7 | sqlite 1 2 3 4... | OpenSSL 1 2 3 4 5 6 7 |
| LibreOffice 1 2 3 4 | poppler 1 2... | freetype 1 2 |
| GnuTLS 1 | GnuPG 1 2 3 4 | OpenSSH 1 2 3 4 5 |
| PuTTY 1 2 | ntpd 1 2 | nginx 1 2 3 |
| bash (post-Shellshock) 1 2 | tcpdump 1 2 3 4 5 6 7 8 9 | JavaScriptCore 1 2 3 4 |
| pdfium 1 2 | ffmpeg 1 2 3 4 5 | libmatroska 1 |
| libarchive 1 2 3 4 5 6 ... | wireshark 1 2 3 | ImageMagick 1 2 3 4 5 6 7 8 9 ... |
| BIND 1 2 3 ... | QEMU 1 2 | lcms 1 |
| Oracle BerkeleyDB 1 2 | Android / libstagefright 1 2 | iOS / ImageIO 1 |
| FLAC audio library 1 2 | libsndfile 1 2 3 4 | less / lesspipe 1 2 3 |
| strings (+ related tools) 1 2 3 4 5 6 7 | file 1 2 3 4 | dpkg 1 2 |

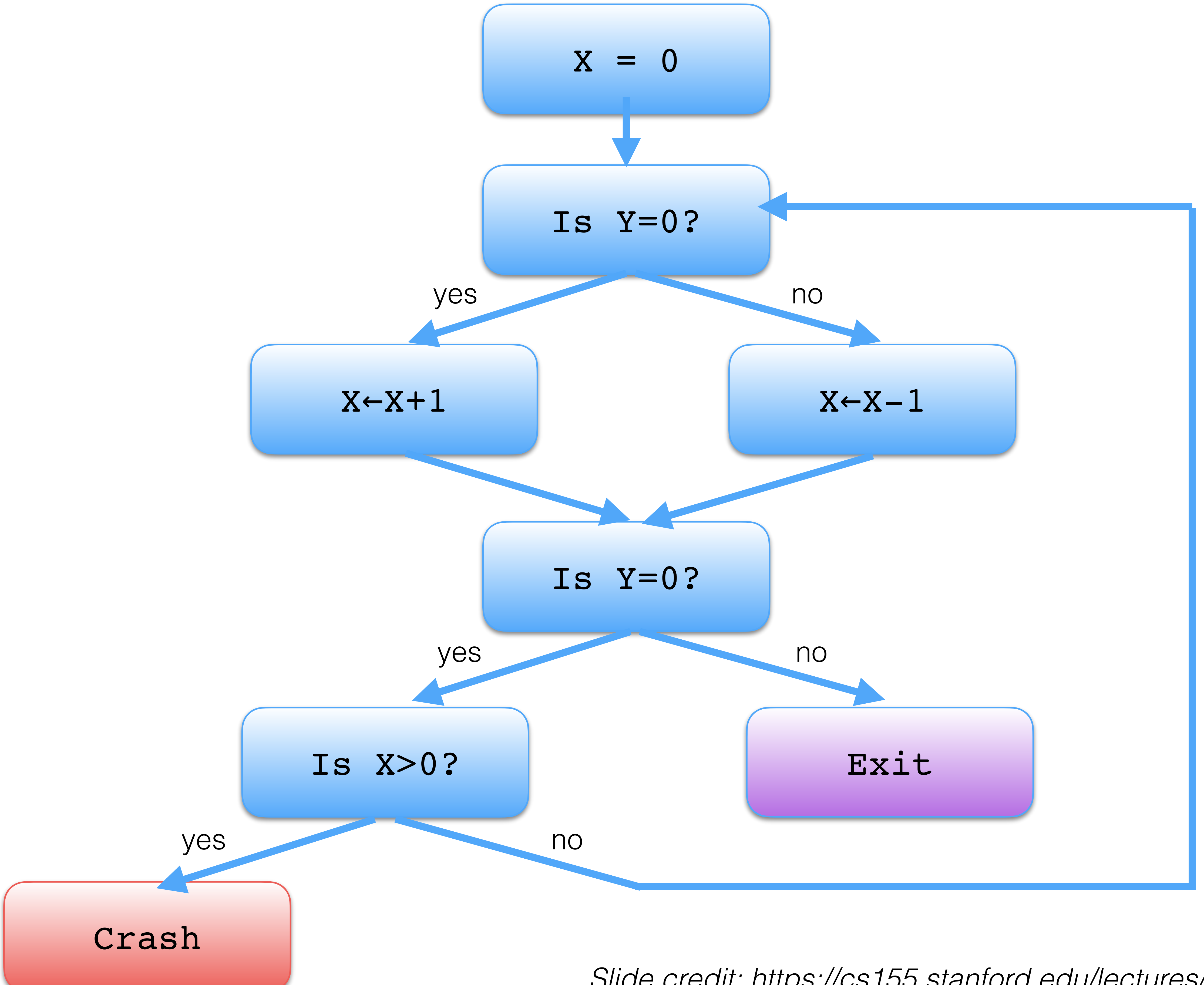
# Symbolic Execution

- Instead of actually running program, track variables as abstract symbols.
- Emulate running program, adding constraints on variables.
- Check algebraically for a solution to assign values and cause crash.

**Pros**: Get an automated proof that code is correct.

**Cons**: Usually only works on small pieces of code. State space explodes exponentially.

- Solve if there exists input Y causing crash.




*Slide credit: https://cs155.stanford.edu/lectures/06-testing.pdf*

The End