CENSUS

IT Security Works

USING PROGRAM INSTRUMENTATION TO IDENTIFY SECURITY BUGS

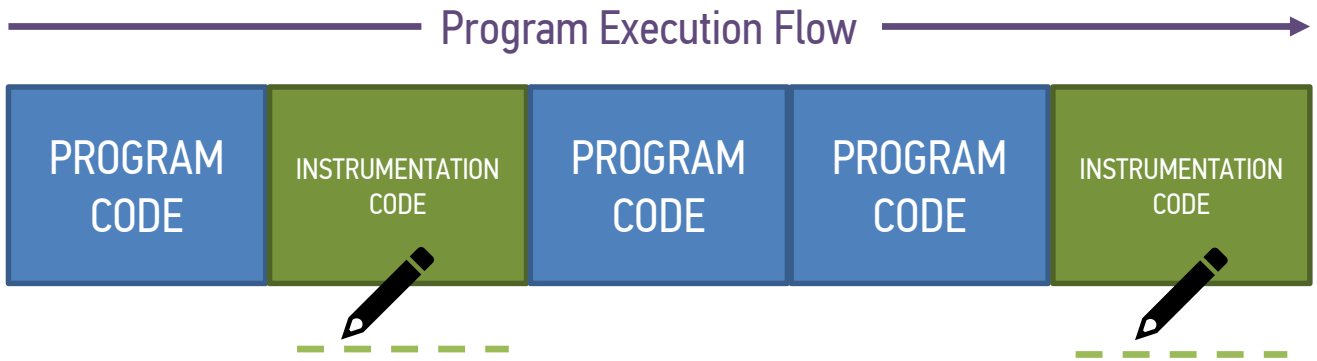
DIMITRIOS GLYNOS (@dfunc on twitter)
dimitris@census-labs.com

Oπe\n conf 2020

www.census-labs.com

> WHAT IS PROGRAM INSTRUMENTATION?

- The process of adding special instructions to a program in order to:
  - Monitor / measure its performance
  - Diagnose errors
  - Write trace information



> ABOUT THIS PRESENTATION

- Can program instrumentation help us find *security bugs?*
- Bugs that someone may use to:
  - ***terminate*** the execution of a program
  - ***alter*** the execution of a program
  - ***retrieve*** program secrets

> PROGRAM ANALYSIS

- Program Analysis
  - Automated reasoning about program semantics
  - *"Are there potential buffer overflows in this program?"*
  - Reasoning is sometimes hard; but instrumentation can help!
- Static vs Dynamic Analysis Techniques
  - *Static analysis:* program analysis for software ***at rest***
  - *Dynamic analysis:* program analysis for software ***during execution***

> INSTRUMENTATION & PROGRAM ANALYSIS

	Access to Source Code	Access Only to Binary
Static Analysis	(1) Annotations	?
Dynamic Analysis	(2) Compile-time instrumentation	(3) Static Binary Rewrite (4) Dynamic Binary Instrumentation (DBI)

> ANNOTATIONS TO AID STATIC ANALYSIS

> ANNOTATIONS FOR CLANG ANALYZER

```
1  int bar(int*p, int q, int *r) __attribute__((nonnull(1,3)));
2
3  int foo(int *p, int *q) {
4     return !p ? bar(q, 2, p)
                  : bar(p, 2, q);
6  }
```

1 '?' condition is true

2 Null pointer passed as an argument to a 'nonnull' parameter

From https://clang-analyzer.llvm.org/annotations.html#attr_nonnull

> COMPILE-TIME INSTRUMENTATION

> GOOGLE SANITIZERS

- Instrumentation to **terminate** application and **report issue** when an undesired condition occurs
  - *AddressSanitizer* – Memory Corruption detector
  - *ThreadSanitizer* – Data Race detector
  - *KCSAN* – Kernel Data Race detector
  - …

> AddressSanitizer CRASH REPORTING

- AddressSanitizer provides stack unwinding and other terse reporting to aid root cause analysis

```
 $ ./buggy-program-compiled-with-asan afl_outputs/crash_input_001
==74917==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60b00000aff4 at pc
0x0000004008dc bp 0x7ffdb826d790 sp 0x7ffdb826d780
WRITE of size 1 at 0x60b00000aff4 thread T0
 #0 0x4008db in offbyone (/home/f/afl/buggy-program-compiled-with-asan+0x4008db)
    #1 0x400927 in main (/home/f/afl/buggy-program-compiled-with-asan+0x400927)
…
0x60b00000aff4 is located 0 bytes to the right of 100-byte region
[0x60b00000af90,0x60b00000aff4)
allocated by thread T0 here:
    #0 0x7fa01eafc602 in malloc (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x98602)
    #1 0x40089b in offbyone (/home/f/afl/buggy-program-compiled-with-asan+0x40089b)
SUMMARY: AddressSanitizer: heap-buffer-overflow ??:0 offbyone
```


> AddressSanitizer MEMORY AFTER OFF-BY-ONE

```
Shadow bytes around the buggy address:
  0x0c167fff95a0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c167fff95b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c167fff95c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c167fff95d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c167fff95e0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c167fff95f0: fa fa 00 00 00 00 00 00 00 00 00 00 00[04]fa
  0x0c167fff9600: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c167fff9610: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c167fff9620: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c167fff9630: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c167fff9640: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```


> BUT HOW DO WE TRIGGER A SANITIZER?

- Need to “drive” code to interesting parts
  - Use Unit Testing
  - Use Fuzzing!
- **Fuzzing**: *black box* testing technique *“for discovering faults in software by providing unexpected inputs and monitoring for exceptions”*

> IMPORTANT FEATURES OF A FUZZER

- Coverage driven
  - Find inputs that exercise as many paths as possible
  - Coverage stats gathered through instrumentation!
- Context aware
  - Some paths are hard to reach (even with a solver), e.g. branch where a CRC was found to be correct
  - Create appropriate input for the specific file format / protocol being tested

> AFL DEMO

> BEYOND C/C++ FUZZING

- Fuzzing using coverage instrumentation, but in other languages
  - go-fuzz (fuzzing Go code)
  - jsfuzz (fuzzing JavaScript code)
  - JQF (fuzzing JAVA code)
  - SharpFuzz (fuzzing .NET IL code)
  - ...

> PROTECTING PRODUCTION BINARIES

- Can we use instrumentation to stop an attack?
  - Google Sanitizers are too elaborate (read: slow) for production binaries…
  - However, we can use:
    - ***Canary stack protection*** – crash when "canary" guard gets overwritten
      - See `-fstack-protector` option
    - ***Control Flow Integrity (CFI)*** – crash if function was called out of context
      - See `-fsanitize=cfi*` (Clang) or `-fcf-protection` (GCC on Intel)
    - ***Pointer Authentication for ARM*** – crash if pointer value was not created by the program
      - See `-mbranch-protection` and `-msign-return-address`

> CFI DEMO

> BINARY INSTRUMENTATION

> STATIC BINARY REWRITE

- Injecting instrumentation into a binary and keeping the binary sound is a non-trivial task
- Why do this?
  - Inject security controls (e.g. stack canaries) to 3rd party blob
  - Reassemble binary so that public exploits won't work
  - Enable coverage guided fuzzing
  - Adding google-sanitizer-like instrumentation
- Many frameworks with growing architecture support:
  - McSema
  - MIASM
  - multiverse

> DYNAMIC BINARY INSTRUMENTATION

- Inject instrumentation while program executes
  - Get binary rewriting benefits without touching the binary...
  - Userspace-level injection
    - Valgrind (ready-made recipes for memory checks etc.)
    - DynamoRIO (framework for injecting instrumentation)
    - Intel Pin (not FOSS, but excellent support for Intel arch.)
  - Virtualization-level injection
    - See AFL QEMU mode

> VALGRIND DEMO

> CONCLUSIONS

> INTEGRATING INSTRUMENTATION IN THE SDLC

Development

- Annotation-type guidance of Static Analysis

Testing

- Dynamic Analysis during Unit Testing
- Focused Fuzzing Campaigns
- Focused Closed Source Component Inspection using DBI

Production

- Stack Canaries
- Pointer Authentication
- CFI

> FOR MORE INFORMATION

- See our FOSSCOMM 2018 presentations on *"Instrumentation With and Without Source Code"* (they cover much more than just security uses!)
- *"Fuzzing: Brute Force Vulnerability Discovery"*, by Sutton, Greene and Amini
- *"From hack to elaborate technique - A survey on binary rewriting"*, by Wenzl, Merzdovnik, Ullrich and Weippl
- https://github.com/google/sanitizers
- https://lcamtuf.coredump.cx/afl/
- https://valgrind.org
- https://github.com/DynamoRIO/dynamorio
- https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html

Thank you!

CENSUS

IT Security Works