CENSUS
IT Security Works

PROGRAM INSTRUMENTATION WITH SOURCE CODE

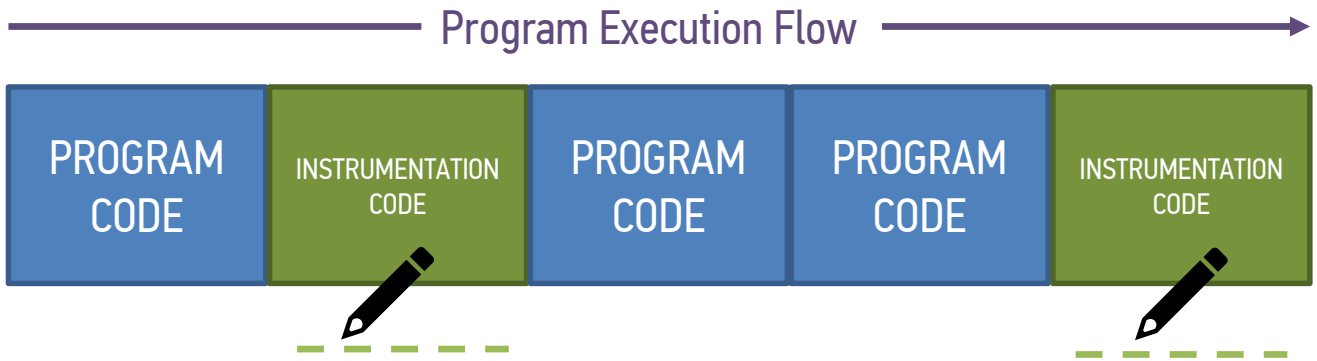
DIMITRIOS GLYNOS (@dfunc)
dimitris@census-labs.com

FOSSCOMM 2018 HERAKLION, CRETE

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



> IT'S AN EXPERIMENT!

- Much like experiments in real life
  - Fact #1: One can only measure if the measurement code is executed
    - Program may need to be *driven* to those points
  - Fact #2: The program may not behave in the same way when instrumented
    - Need for lightweight / unobtrusive instrumentation

> EXAMPLE USES

- Many domains
  - Debugging
  - Tracing
  - Profiling for program optimization
  - Memory management error detection
  - Threading error detection
  - Code coverage measurements (for testing etc.)

> A CRUDE EXAMPLE

Code:

```
int add(int a, int b) {
        /* instrumentation code */
       fprintf(stderr, "%s() called at %ld\n",
                __FUNCTION__,
                time(NULL));
       return a + b;
}
```

Instrumentation output:

```
add() called at 1537547793
```


> INSTRUMENTATION FLAVOURS

- Source Code vs Binary Instrumentation
  - Will the instrumentation code have access to source-level semantics?
- Static vs Dynamic Analysis Techniques
  - *Static analysis* is program analysis for software ***at rest***
  - *Dynamic analysis* is program analysis for software ***during execution***

> STATIC-ONLY INSTRUMENTATION?

- Modifying the code to aid static analysis
- Not very common
  - Most instrumentation use cases employ dynamic analysis at some point
  - Intermediate Language level information is usually sufficient for static analysis purposes
- How about “marking” parts of the code?
  - Compile-time checks based on *attributes* or *annotations*

> ANNOTATION ASSISTED STATIC ANALYSIS

```
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

> INSTRUMENTATION FOR DEBUGGING

- Compilers do not normally emit instrumentation code for debugging purposes
  - Options **–g** and **–ggdb** in GCC add debugging ***data***
- Such instrumentation code is usually inserted
  - by the developer (e.g. print stack information through *libunwind*)
  - by the debugger (in-memory program rewriting)

> LIBUNWIND

- A portable stack unwinding C API.
  - Get call chain
  - Get function name and start / end addresses
  - Are we executing signal code?
  - Examine register contents
  - Perform operations within the execution context of another process (ptrace)
  - https://www.nongnu.org/libunwind/

> LIBUNWIND EXAMPLE

Example code from https://github.com/daniel-thompson/libunwind-examples/blob/master/unwind-local.c

```
// stack backtrace() using libunwind
int cmp(const void *a, const void *b)
{
    backtrace();
    exit(0);
}

int main(int argc, char **argv)
{
    int data[] = {1, 2, 3, 4};
    int needle = 4;
    bsearch(&needle, data,
            lengthof(data),
            sizeof(data[0]), cmp);
    return 1;
}
```

```
$ ./unwind-local
0x400ba9: (cmp+0x8)
0x7f09f4341207: (bsearch+0x57)
0x40096b: (main+0x5b)
0x7f09f432a830:
(__libc_start_main+0xf0)
0x4009b9: (_start+0x29)
0x0: -- no symbol name found
```


> INSTRUMENTATION FOR CODE COVERAGE

- Tracking code flow for (unit) testing purposes
  - Track which parts of the code were exercised
  - See *gcov* (GCC) / *SanitizerCoverage* (LLVM, plugins!)
- Profiling
  - Measure the number of calls made to a function
  - Measure where the program spent significant time
  - Perform optimizations based on the above data
  - See *gprof* (binutils)
- Hardware Assisted Branch Tracing
  - See *perf* + Intel PT (5th generation Intel Core or better)

> GCOV INSTRUMENTATION EXAMPLE

```
# installing instrumentation and generating graph data (prog.gcno)
$ gcc –fprofile-arcs –ftest-coverage –o prog prog.c
```

```
# Vanilla version
mov    %rsi,-0x10(%rbp)
mov    $0x400d78,%edi
callq  400720 <puts@plt>
```

```
# Instrumented Version
mov    %rsi,-0x10(%rbp)
mov    0x2043a3(%rip),%rax
# 6053e0 <__gcov0.print_stat_info>
add    $0x1,%rax
mov    %rax,0x204398(%rip)
# 6053e0 <__gcov0.print_stat_info>
mov    $0x403868,%edi
callq  400d40 <puts@plt>
```

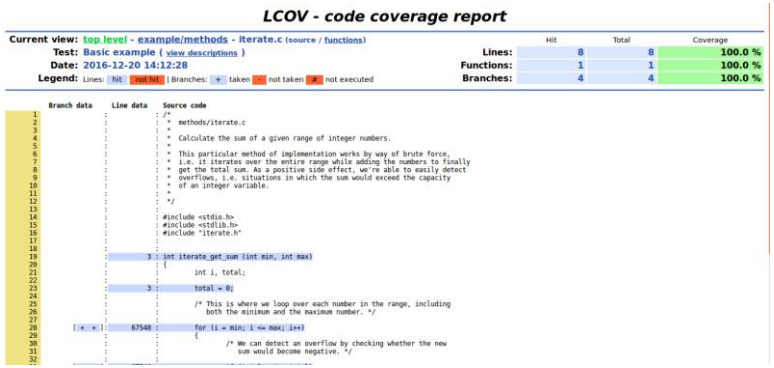
- Each instrumented point raises a counter and results are flushed to a data file at program exit (prog.gcda)

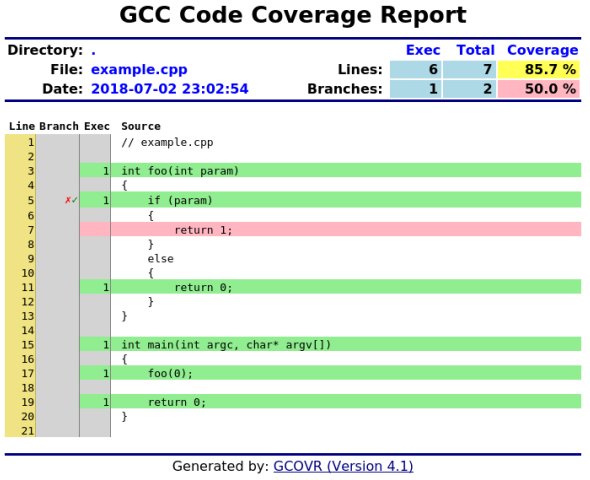
> GCOV INSTRUMENTATION EXAMPLE

```
# run the software to generate coverage data (prog.gcda)
$ ./prog <args>
# merge graph and measurement data in one report with gcov (prog.c.gcov)
$ gcov prog
File ‘prog.c'
Lines executed:67.44% of 43
Creating ‘prog.c.gcov’
$ cat prog.c.gcov
…
 1706: 37:switch (S_IFMT & (stat_buffer->st_mode)){
 1605: 38:        case S_IFREG: printf("Filetype: Regular File\n"); break;
#####: 39:        case S_IFSOCK: printf("Filetype: Socket\n"); break;
…
# Line 37 got 1706 invocations, Line 38 got 1605 while Line 39 got none.
```


> TRANSFORMING GCOV OUTPUT

- See *lcov* and *gcovr* projects for
  - Alternate output formats (HTML, XML etc.)
  - Integration with Continuous Integration environments
    - Jenkins
    - Travis CI





> CODE COVERAGE AND FUZZING

- **Fuzzing** (or Fuzz Testing) is a *black box* security testing technique *"for discovering faults in software by providing unexpected inputs and monitoring for exceptions"*
  - *Fuzzing*, M. Sutton, A. Greene, P. Amini, Addison Wesley, 2007
- Stack unwinding instrumentation provides fuzzers with info about the root cause of a fault
  - if the same bug was triggered twice, record it once
- Code coverage instrumentation provides fuzzers with information about unexplored paths

> THE AFL FUZZER

- American fuzzy lop (AFL)
  - Popular fuzzer that employs compile-time instrumentation and genetic algorithms to automatically generate test cases that trigger new states in the tested program
  - http://lcamtuf.coredump.cx/afl/
  - Has identified an impressive number of memory corruption vulnerabilities in common FOSS software

> AFL INSTRUMENTATION IN A NUTSHELL

- Instruments entry point, branch labels and conditional branching at assembler level
- Instrumentation records *directed branches* and hit counts
- A parent program (*afl-fuzz*) provides mutated inputs to the target program running in a child process
  - No need to *execve()* for each input, the instrumented child listens for parent commands and handles input in forked process (see *forkserver*)
  - Instrumentation data are shared with parent via a Shared Memory segment
  - Mutated inputs that discovered new “branches” are added to the input queue (they do not replace existing inputs)

> AFL DEMO

> IDENTIFYING MEMORY ACCESS ERRORS

- *AddressSanitizer*
  - Instrumentation added by compiler to detect invalid memory access errors
  - Supported by LLVM and GCC
  - Detects use after free issues, buffer overflows, etc.
  - Instrumentation uses a “shadow” copy of memory for book keeping purposes
  - A runtime library (*libasan*) replaces malloc & free

> IDENTIFYING MEMORY ACCESS ERRORS

- AddressSanitizer Algorithm
  - "redzones" are memory regions with marked bytes used to track overwrites
  - "redzones" are noted as "poisoned" regions
  - Any "poisoned" region accessed is reported and leads to abort()

Instrumentation prefix

```
if (IsPoisoned(address))
    {
      ReportError(address,
    kAccessSize, kIsWrite);
    }
```

Original Code

```
*address = ...;
```

OR

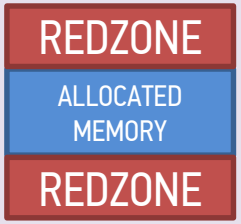
```
... = *address;
```

Stack Buffer Protection

Variables are intermingled with poisoned redzones

Redzones are unpoisoned at function exit

Heap Buffer Protection



malloc() returns regions surrounded by poisoned redzones

free() poisons freed region and delays its reallocation

> WHAT DRIVES AddressSanitizer?

- AddressSanitizer will need something to drive the target application into interesting code paths
  - Unit Testing? A Fuzzer?

```
 $ ./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 REDZONE 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
```


> OTHER SANITIZERS

- Other interesting sanitizers are:
  - *ThreadSanitizer*
    - data race detector (GCC + LLVM)
  - *MemorySanitizer*
    - uninitialized memory read detector (LLVM only)
  - *UndefinedBehaviourSanitizer*
    - Catches undefined behavior (GCC + LLVM)
  - *LeakSanitizer*
    - Memory leak detector (GCC + LLVM)

> INSTRUMENTATION TO HINDER BUG EXPLOITATION

- Idea: detect bug exploitation & terminate application
  - Stack Protector (GCC / LLVM)
    - Adds “canary” value between local variables and saved frame pointer
    - moves function pointer variables before others
    - Abort() if “canary” value is overwritten at function exit through (presumably) a stack buffer overflow
  - Shadow Call Stack (Experimental, LLVM)
    - Keep function’s return address in “shadow” memory and checks prior to function exit
  - CFI (protects statically linked part of code, LLVM)
    - Build control flow graph and check via instrumentation if execution will be transferred to whitelisted points in the code

> KERNEL INSTRUMENTATION

- kprobe / uprobe
  - Live modification of kernel & userspace code (by the kernel) for instrumentation purposes
- tracepoints / LTTng
  - tracepoints for significant events in Linux kernel code
- DTrace / SystemTap / bpftrace / perf
  - Configure trace-points, collect trace information and inject code (where applicable)
- eBPF
  - An in-kernel virtual machine for dynamic code execution

> KERNEL FUZZING

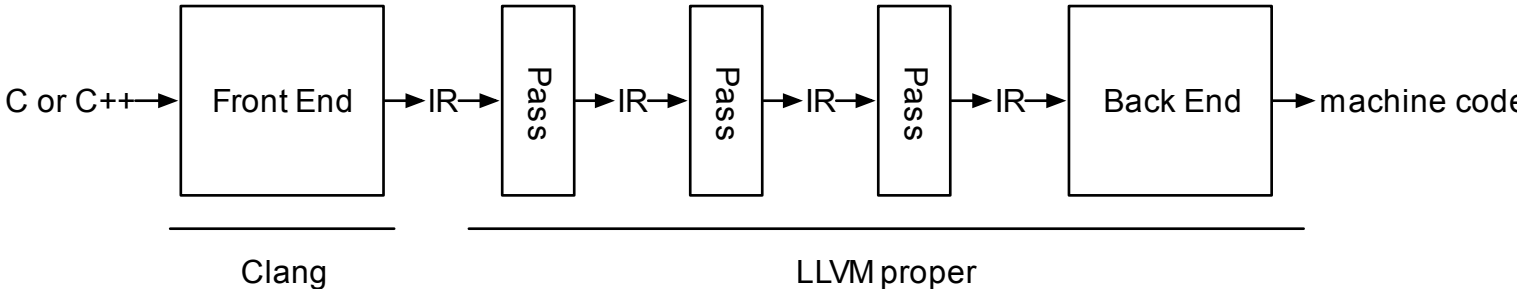
- syzkaller
  - An unsupervised coverage-guided kernel (system call) fuzzer
  - https://github.com/google/syzkaller
- Kernel CONFIG_KCOV option
  - Required by syzkaller
  - Works in conjunction with -fsanitize-coverage=trace-pc compiler instrumentation
  - exposes kernel code coverage information in a form suitable for coverage-guided fuzzing
- Kernel CONFIG_KASAN option
  - AddressSanitizer for the Linux Kernel (x86_64 & ARM64)

> BUILDING YOUR OWN INSTRUMENTATION

- Take into consideration
  - Speed requirements
  - Threading requirements (reentrant code)
  - Memory usage requirements
  - Data storage requirements
- Is there something like an instrumentation standard?
  - Yes, see OpenGroup's "Application Response Measurement" (ARM) standard
- What's a portable way to apply transformations to code?
  - See LLVM IR Pass code

> LLVM IR PASS

- LLVM allows to run code transforms as an Intermediate Representation (IR) Pass



Source: https://www.cs.cornell.edu/~asampson/blog/llvm.html

> INJECTING A LLVM IR PASS

```
$ git clone https://github.com/sampsyo/llvm-pass-skeleton.git
$ cat llvm-pass-skeleton/skeleton/Skeleton.cpp
…
virtual bool runOnFunction(Function &F) {
  errs() << "I saw a function called " << F.getName() << "!\n";
  return false;
}
…
$ cd llvm-pass-skeleton
$ mkdir build
$ cd build
$ cmake ..
$ make
$ clang -Xclang -load -Xclang build/skeleton/libSkeletonPass.* something.c
I saw a function called main!
```

Source: https://www.cs.cornell.edu/~asampson/blog/llvm.html

> REFERENCES

- https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
- https://sourceware.org/binutils/docs/gprof/
- https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing
- https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/intel-pt.txt
- http://halobates.de/blog/p/410
- https://www.amazon.com/Fuzzing-Brute-Force-Vulnerability-Discovery/dp/0321446119
- http://www.brendangregg.com/blog/2018-10-08/dtrace-for-linux-2018.html
- https://gcc.gnu.org/onlinedocs/gcc-8.2.0/gcc.pdf
- https://clang.llvm.org/docs/UsersManual.html
- http://lcamtuf.coredump.cx/afl/technical_details.txt
- https://clang.llvm.org/docs/ShadowCallStack.html
- https://clang.llvm.org/docs/ControlFlowIntegrity.html
- https://collaboration.opengroup.org/tech/management/arm/
- https://www.cs.cornell.edu/~asampson/blog/llvm.html

Thank you!

CENSUS

IT Security Works