Code Coverage

Chris.Rohlf@leafsr.com
@chrisrohlf
leafsr.com
Agenda

- **WHAT** is code coverage?
- **WHY** use code coverage?
- **HOW** can we measure code coverage?
- **Case study:** Fuzzer effectiveness
What is Code Coverage?

• Measuring which lines of code or instructions are executed at runtime when a program processes a specific input
  • Example: unit tests attempt to get code coverage by using functionality in an application or API

• http://en.wikipedia.org/wiki/Code_coverage
Why Code Coverage?

• ... because not all inputs are created equally
Why Code Coverage?

- Usability and Reliability
  - Well tested code will result in fewer bugs
  - “Do our features work? Was this particular line of code run?”

- Security
  - Untrusted inputs will taint data structures and eventually lead to security vulnerabilities in various places throughout the code
  - “Why and how was this particular line of code run?”
  - “What are the potential code paths to this vulnerable function?”
How can we use Code Coverage?

• Traditional code coverage is focused on *which features did we exercise? Did the output match what was expected?*

• Security code coverage is generally focused on *which features did we exercise with what set of inputs? What was the result of the runtime?*

  • There could be multiple paths to vulnerable code

  • We are concerned with the edge cases, the unexpected inputs and challenging developer assumptions about program behavior
Code Coverage

• Example code coverage output while fuzzing:

  • **Input #1:** <binary file abc.bin, modified offsets 0x4, 0x10, 0x23ed>
    Functions hit: 20/1504
      Function 1: sub_104c908
        Hit 1: loop at 0x104c9b2 iterated 12 times
        Hit 2: loop at 0x104c9b2 iterated 6 times
      Function 2: sub_105f00b
        Hit 1: loop at 0x105f012 iterated 12 times
        Hit 2: loop at 0x105f012 iterated 6 times

  • **Input #2:** <binary file abc.bin, modified offsets 0xc, 0xf, 0x1c>
    Functions hit: 513/1504
      ...
Code Coverage Gotchas

- Application configurations will change
- Non-deterministic program states
  - Heap pressure, threads (race conditions)
  - These will inevitably bite you and feel like non-deterministic behavior
Source Code Coverage

- Automated source code coverage requires support at compilation time
  - Source level debugging requires additional meta-data is compiled into the target
  - http://llvm.org/docs/SourceLevelDebugging.html
Source Code Coverage

• gcov is an open source tool for source level code coverage analysis and profiling

• Produces an annotated version of the source file based on runtime logging of basic block execution
Code Coverage

- Code coverage is easy to do with source code access

- Code coverage of binary-only programs requires a specialized toolset
  - Start with zero knowledge
  - Automation
  - Eventually needs to scale with program size and complexity
    - Custom components (memory allocators etc.)
Dynamic Code Coverage

- Static analysis with IDA Pro can be used to provide a list of interesting code locations to tell our dynamic tools about
  - All functions and basic blocks
  - But it can’t tell us everything about indirect calls (such as those that are vtable dispatched)
  - Can tell us what code exists, not what it does or what inputs are required to reach it
Dynamic Code Coverage

• Hit tracing

  • Specify which instructions or functions in the binary you want to monitor for execution

  • Trigger a callback each time they are executed, log some output (function arguments, caller etc.), then continue execution
Function Tracing

- Function level tracing
  - Record which functions are executed for a given input
  - The order of function execution, call paths between function X and function Y
  - Function return values
  - Does not measure code coverage very deeply but normally faster
Basic Block Tracing

• Basic Block level tracing
  • A basic block is defined as a chunk of code with one entry point and one exit point
  • Measuring code coverage at the basic block level produces detailed results but can be slower
  • http://en.wikipedia.org/wiki/Basic_block
Basic Block Tracing

- Basic blocks of x86 instructions
Basic Block Tracing

- Basic block level tracing is granular enough for most simple code coverage efforts but has some blind spots
  - Comparison operations such as strcmp/memcmp may be reduced to a single instruction: `repz cmps`
  - Simple, but important, arithmetic logic may also be hidden inside a basic block
Debugger Based Coverage

- Software based breakpoints on the x86
  1. Read and store original instruction at offset X
  2. Replace byte/instruction at offset X with 0xCC (int3 instruction)
  3. Catch debugger event, inspect program state
  4. Replace 0xCC with original byte/instruction, single step the process
  5. Replace original byte/instruction with 0xCC, continue the process
Debugger Based Coverage

• Software based breakpoints

• Gives us opportunity to inspect the program at runtime at a predetermined execution point through a (usually) well defined and supported API

• What do inputs look like? State of internal data structures? Memory usage?

• Slow, does not scale to hundreds of thousands of breakpoints due to constant context switching

• http://gdtr.wordpress.com/2012/05/11/fuzzing-hit-tracing/
Dynamic Binary Instrumentation

• Interacting with a binary at runtime via injecting instrumentation code using JIT techniques
  • Code rewriting, patched jumps to instrumentation code before and after original code
  • No breakpoints, no debugger traps/events: faster

• Intel Pintool, closed source but feature-rich and well documented API
Dynamic Binary Instrumentation

- Pintool loads up your custom C/C++ dll/so and instruments the target application as requested

```c
// size = size of buffer requested
// from = instruction address malloc was called from
void mallocBefore(ADDRINT size, ADDRINT from) {
    ...
}

// ret = location malloc returned in the heap
void mallocAfter(ADDRINT ret) {
    ...
}
```

- Invoked every time a new image is loaded into the process

```c
VOID Image(IMG img, VOID *v) {
// Check if this image implements malloc()
    RTN mallocRtn = RTN_FindByName(img, "malloc");

    if(RTN_Valid(mallocRtn)) {
        RTN_Open(mallocRtn);
        RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR) mallocBefore,
                       IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_RETURN_IP,
                       IARG_END);
        RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR) mallocAfter,
                       IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);
        RTN_Close(mallocRtn);
    }
```
Dynamic Binary Instrumentation

- Function and basic block hooking is also possible
- Execute a callback for each one and record function arguments, register states and more
- Hook all image (DLL/SO) loads
VOID Trace(TRACE trace, VOID *v) {
    for(BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl)) {
        if(KnobFTrack) {
            INS tail = BBL_InsTail(bbl); // Last instruction in the basic block

            // Track function calls
            if(INS_IsCall(tail)) {
                // Direct calls
                if(INS_IsDirectBranchOrCall(tail)) {
                    ADDRINT t = INS_DirectBranchOrCallTargetAddress(tail);
                    INS_InsertPredicatedCall(tail, IPOINT_BEFORE, AFUNPTR(LogCallAndArgs),
                                            IARG_ADDRINT, t, IARG_FUNCARG_ENTRYPOINT_VALUE,
                                            0, IARG_FUNCARG_ENTRYPOINT_VALUE, 1, IARG_FUNCARG_ENTRYPOINT_VALUE,
                                            2, IARG_END);
                } else {
                    // Indirect calls
                    INS_InsertCall(tail, IPOINT_BEFORE, AFUNPTR(LogIndirectCallAndArgs),
                                   IARG_BRANCH_TARGET_ADDR, IARG_BRANCH_TAKEN,
                                   IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_FUNCARG_ENTRYPOINT_VALUE, 1,
                                   IARG_FUNCARG_ENTRYPOINT_VALUE, 2, IARG_END);
                }
            }
        }
    }
}

int main(int argc, char *argv[]) {
    ...
    TRACE_AddInstrumentFunction(Trace, 0);
    ...
}
VOID PIN_FAST_ANALYSIS_CALL count_bbl(ADDRINT instc, ADDRINT addr) {
    fprintf(LogFile, "%x", addr);
}

void LogCallAndArgs(ADDRINT ip, ADDRINT arg0, ADDRINT arg1, ADDRINT arg2) {
    if(debug)
        fprintf(LogFile, "%s : %x(%x, %x, %x)\n", i->name, ip, arg0, arg1, arg2);
}

void LogIndirectCallAndArgs(ADDRINT target, BOOL taken, ADDRINT arg0, ADDRINT arg1, ADDRINT arg2) {
    if(!taken)
        return;

    LogCallAndArgs(target, arg0, arg1, arg2);
}
Case Study: Fuzzing

- Fuzzer effectiveness
  - Monitoring inputs
  - Program state and memory conditions
- “Corpus Distillation”
Case Study: Fuzzing

- Mature applications have tackled their code coverage and reliability testing via unit-tests
  - These are a great place to start when fuzzing as they are guaranteed to reach some functionality
  - Mutate unit-tests to reach interesting code paths
Case Study: Fuzzing

- Dumb fuzzing used to be /dev/urandom but today even the dumbest fuzzers bit flip existing inputs
  - You need some understanding of the protocol/format
- Smart fuzzers are often capable of generating inputs according to a format or specification
Case Study: Fuzzing

- Program State

- There are some non-deterministic behaviors that may differ between different executions
  - Heap state: how much memory is currently allocated? How often is garbage-collection occurring?
  - Threads: how many threads are concurrently running? What resources are exclusive and which are shared?
Case Study: Fuzzing

- Monitoring Inputs

- Specialized pintool for tracing program behavior
  - IDA basic block addresses ⟷ pintool ⟷ pin callbacks ⟷ fuzzer / IDA

- Pintool attempts to answer some key questions:
  - How many basic blocks are being hit?
  - What % of blocks are being hit multiple times?
  - How many different code paths are being hit from a specific branch/path?
Case Study: Fuzzing

- Monitoring Inputs

- Automated fuzzer feedback to improve code coverage is not a solved problem just yet
  - Requires monitoring the execution of basic blocks you and determining why you aren’t hitting them via constraint solving...
Case Study: Fuzzing

• Corpus Distillation

• Google did some great research in fuzzer code coverage

• “Eliminates the requirement for protocol grammar via automated observation of the software to be tested”

• Start with thousands of input samples, distill them down to the ones that reach an acceptable level of code coverage
Case Study: Fuzzing

- Start with an input set \( Y \) and a target \( X \)

- Trace all basic block executions in \( X \) while processing inputs \( Y_1 \ldots Y_N \)

- Diff the trace outputs to determine what the minimum required set of those samples is to reach the desired code coverage
Case Study: Fuzzing

- Smart Fuzzing: making your fuzzer protocol/format aware, use constraint solving to improve inputs

- Sub-instruction profiling “using sub-instruction profiling, essentially using DBI to instrument common code patterns that may shield hidden logic”
  - String comparisons, checksums
  - No complex constraint solving required