PowerFL:
Fuzzing VxWorks embedded systems

Peter Goodman
Artem Dinaburg
Trent Brunson

Introductions

Peter Goodman

Senior Security Engineer

peter@trailofbits.com

Artem Dinaburg

Principal Security Engineer

artem@trailofbits.com

Trent Brunson

Director of R&D

trent.brunson@trailofbits.com

PowerFL: A VxWorks bug-finding capability

- Combines the AFL fuzzer with the QEMU virtual machine to fuzz PowerPC and Intel i386 VxWorks targets on commodity computers

PowerPC™ + AFL = PowerFL

- Approach generalizes beyond VxWorks (e.g. to automotive and SCADA systems)

It vxworks, but it's not magic!

- **We developed a *prototype* that proves that semi-automated bug-finding for embedded VxWorks targets is feasible**
  - It is *not* a production quality bug finding powerhouse
- **Based on proven technologies (AFL fuzzer, QEMU)**
- **Requires varying levels of manual setup and analysis depending on the target**
  - Most targets *won't* work out of the box

Automated bug finding: fact or fiction?

- **DARPA Cyber Grand Challenge pitted machines against machines to automatically find, exploit, and patch bugs**
  - CGC avoided the problem of figuring out how to run the program, how/where the program reads input, etc.
  - Real world programs are much more varied and embedded systems (e.g. VxWorks) are a nightmare of variety
- **Can we generalize CGC systems to real programs?**

From CGC to PowerFL: Embedded systems

- **CGC similarities**
  - Mostly programmed in C and assembly, often implement POSIX-like I/O
  - Distributed as one or two self-contained programs/executables
- **Real-world differences**
  - Variety of hardware (sub)architectures. Will not “just run”.
  - Variety of I/O interfaces, not necessarily well-specified (e.g. MMIO)
  - Variety of input sources, the subset of which are “interesting” from an attacker perspective is *a priori* unknown

Embedded systems of consequence: VxWorks

- **Cars, SCADA and defense platforms run VxWorks**
  - They're system-of-systems, with many individual parts communicating over one or more shared networks
  - Some of this hardware runs old versions of VxWorks
- **Assess and improve security and reliability of physical systems**
  - Hardware may be on a deployment, unique, explosive, or unavailable

VxWorks is a real-time operating system

- **Really just a big program, with lots of `#ifdefs` that configure what components are included**
    - Built from optional components: serial I/O, FTP, NTFS, FAT, etc.
    - Not general-purpose: configured to run on specific hardware, with a known amount of RAM, and a set of number of devices
- **Two common ways of using VxWorks**
    - User program linked against the kernel, included in the kernel image
    - User program downloaded over the network (e.g. FTP), or read from the local file system

AFL is a fuzzer

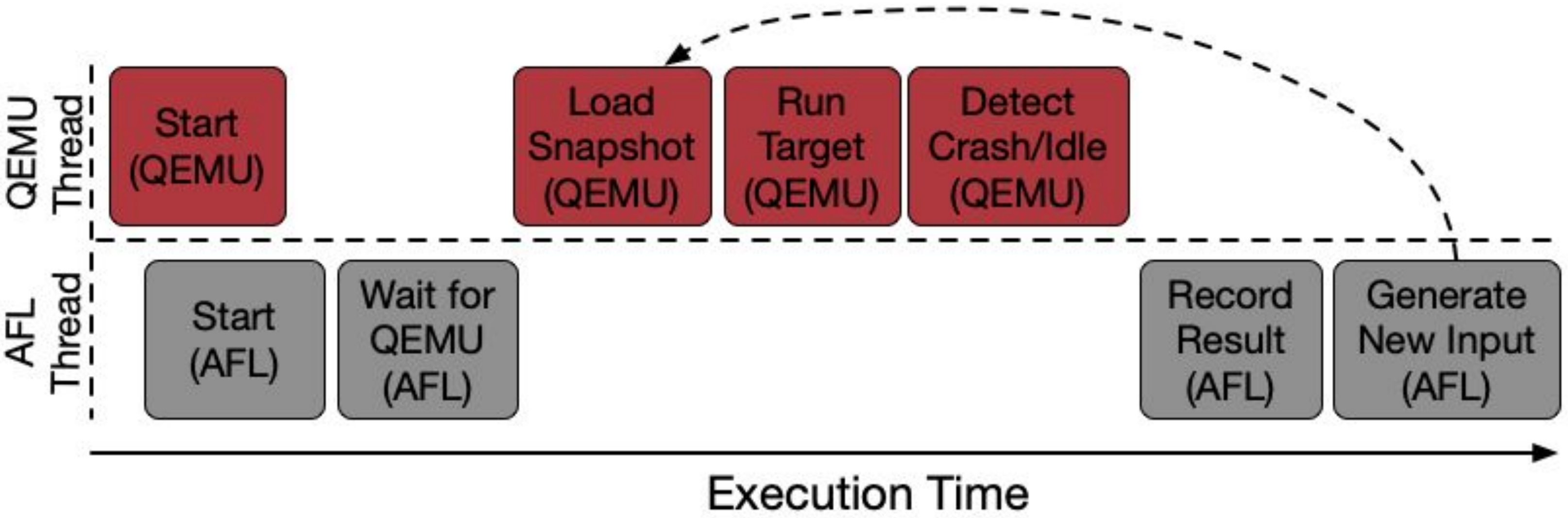
- **AFL generates mutated program inputs and determines whether the new input triggers a bug in the target**
- **AFL is effectively a genetic algorithm that searches through the set of all possible inputs**
  - Code coverage is the fitness function, various mutation operators
- **AFL works on source-available user-mode programs**
  - VxWorks meets *neither* of these requirements
- **Using AFL to fuzz unmodifiable kernel-mode programs?**

QEMU is an emulator

- **QEMU is a whole-system emulator that emulates a wide variety of CPUs and peripherals**
  - Including multiple PowerPC reference boards
- **Uses a common intermediate representation (TCG) to handle a variety of processors.**
  - Instrumentation code is largely portable across processors
  - We can implement code-coverage as a part of the translation process
- **Provides cross-architecture and cross-operating system execution and code coverage**

PowerFL = AFL + QEMU + VxWorks

- **PowerFL can fuzz across architecture and OS boundaries**
  - Novel solutions for i/o passthrough, crash/idle detection, device hooks.



Fuzzing VxWorks: Our incredible journey (0/11)

- **Let's go on a journey to discover how PowerFL works and the rationale behind our design decisions**
  - Provides context for *why* a feature exists, not just how PowerFL works
  - Start with a **goal**, describe the **challenges**, and our **solution**.
  - Each solution generates new **sub-problems**
- **Our decisions were driven by limited resources and the need to rapidly develop a working prototype**
  - We are open to improvements and suggestions

Fuzzing VxWorks: Our incredible journey (1/11)

- **<u>Goal</u>: Fuzz VxWorks PowerPC targets**
- **Challenge: Lack experience with *both* VxWorks RTOS and PowerPC architecture**
- **Solution: Fuzz VxWorks x86 targets, port system to PowerPC when we have a fuzzing capability**
    - We have a lot of experience with the Intel i386 (x86) architecture
    - Mitigated risk by handling only one unknown at a time
    - <u>Bonus</u>: Extra capability: x86 and PowerPC
    - <u>Sub-problem</u>: how do you fuzz VxWorks targets?

Fuzzing VxWorks: Our incredible journey (2/11)

- **Goal**: Run `afl-fuzz` against VxWorks targets
- **Challenge: AFL is a user-mode fuzzer, VxWorks+program execute in supervisor mode**
- **Solution: Emulate VxWorks+program in QEMU, which runs in user mode**
  - AFL embedded into QEMU, runs as separate thread
  - QEMU and AFL threads coordinate their emulation and mutation
  - Sub-problem: VM boot process is deterministic and wastes machine time in a fuzzing campaign

Fuzzing VxWorks: Our incredible journey (2/11)

```
                  american fuzzy lop 2.52b (PowerFL)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 36 sec       │  cycles done : 0      │
│   last new path : none seen yet                      │  total paths : 1      │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 2.42% / 2.42%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : arith 16/8            │ favored paths : 1 (100.00%)            │
│ stage execs : 237/726 (32.64%)      │  new edges on : 1 (100.00%)            │
│ total execs : 903                   │ total crashes : 0 (0 unique)           │
│  exec speed : 25.11/sec (slow!)     │  total tmouts : 0 (0 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/64, 0/63, 0/61                      │    levels : 1          │
│  byte flips : 0/8, 0/7, 0/5                         │   pending : 1          │
│ arithmetics : 0/448, 0/0, 0/0                       │  pend fav : 1          │
│  known ints : 0/0, 0/0, 0/0                         │ own finds : 0          │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 0/0, 0/0                              │ stability : 99.87%     │
│        trim : 0.00%/1, 0.00%                        ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu: 29%]
```

Fuzzing VxWorks: Our incredible journey (3/11)

- **Goal: Run target as fast as possible**
- **Challenge: VxWorks must boot before target executes**
  - Bootloader unpacks and loads VxWorks kernel
  - VxWorks kernel initializes devices and OS state
  - Eventually target program executes
- **Solution: Snapshot VM state when the user program initiates its first I/O operation**
  - Sub-problem: When does the target perform its first I/O operation?

Fuzzing VxWorks: Our incredible journey (4/11)

- **<u>Goal</u>: Interpose on specific guest functions to get "semantic visibility" -- know what the guest is doing**
    - I/O operations, scheduler, exception handlers, device initialization, etc.
- **Challenge: Hook execution at arbitrary points**
- **Solution: Robust function hooking**
    - Hooks injected during QEMU JIT translation
    - Hook function entry points by program counter
    - Hook function exit points by overwriting return addresses on stack
    - <u>Sub-problem</u>: stripped target binaries without symbols

Fuzzing VxWorks: Our incredible journey (5/11)

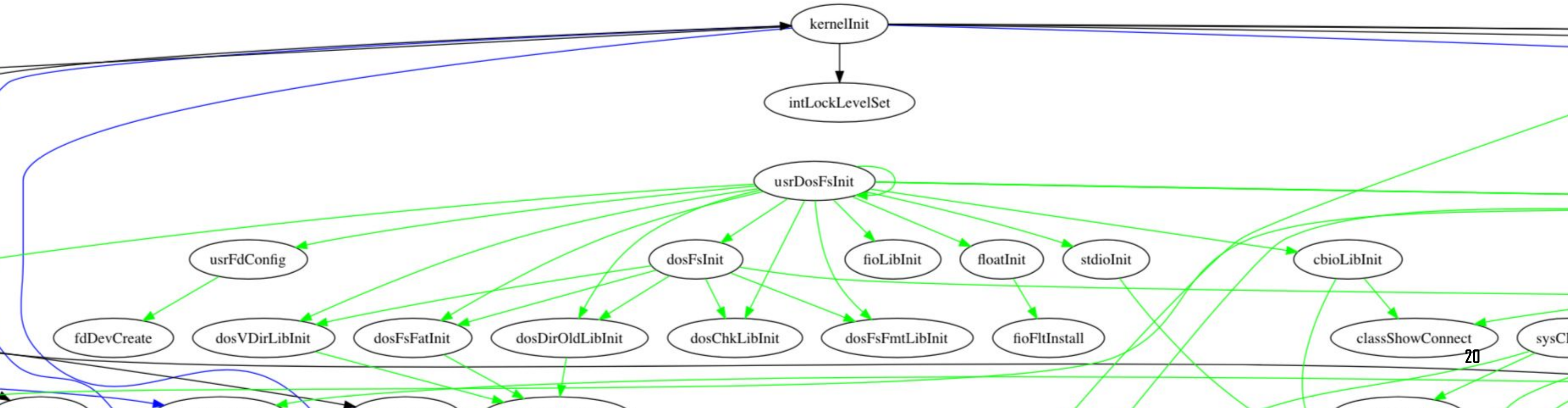
- **<u>Goal</u>: Hook any function by name**
- **Challenge: Stripped binaries without symbol names**
- **Solution: Heuristic function matching**
    - Baseline: Symbolized VxWorks for same architecture, built from source
    - IDA scripts identify functions in stripped binary using info derived from symbolized binary: string cross references, call graph structure, opcode sequences, and FLIRT signatures
    - Mappings saved in symbol file, loaded by PowerFL
    - <u>Caveat</u>: not a 100% solution, some manual effort required

Fuzzing VxWorks: Our incredible journey (6/11)

- **Goal: Support devices/peripherals needed by target**
- **Challenge: Many VxWorks configurations have incomplete QEMU emulation support**
  - Many devices needed by target lack QEMU emulation support
- **Solution: Manually and automatically identify problematic code, stub it out with function hooks**
  - Identified problematic functions can be “stubbed out” by naming those addresses as `powerfl_suppress_N` in symbol map file
  - Sub-problem: Identify functions that might be for device setup

Fuzzing VxWorks: Our incredible journey (7/11)

- **<u>Goal</u>**: Finding what functions to stub in order to “get beyond” initialization of unsupported devices
- **Solution**: Visual diff of function traces, look for callers of `pci`-related functions, function names ending in “`Init`”



Fuzzing VxWorks: Our incredible journey (8/11)

- **<u>Goal</u>: Feed mutated input files from AFL into the target**
- **Challenge: Feeding files from the host into the guest**
  - AFL is a file fuzzer
  - Does the target read input from files? If so, where are they stored?
  - If the the target reads files, then how do we get mutated inputs from the host file system into the guest file system?
- **Solution: Implement transparent file I/O passthrough**
  - Shadow guest file operations into host file system
  - <u>Sub-problem</u>: target program likely doesn't support virtio drivers

Fuzzing VxWorks: Our incredible journey (9/11)

- **Goal**: Transparent (guest unaware) I/O passthrough
- **Challenge:** VxWorks is not general purpose; pre-built binaries not configured with virtual I/O driver support
  - Unlike TriforceAFL, we can't load in our own drivers or programs into the guest
- **Solution:** Hook and translate I/O function effects into "mounted" directory on host
  - Write bytes from host-to-guest on `reads`
  - Read bytes from guest-to-host on `writes`

Fuzzing VxWorks: Our incredible journey (10/11)

- **<u>Goal</u>: Detect if input drove guest to execute new code**
- **Challenge: Interrupts trigger false-positive code coverage**
    - Non-deterministic events that trigger control-flow transfers; don't want these transfers to count for spurious "new" coverage
- **Solution: Instrument JIT-translated guest code, hook interrupt service routines**
    - Block entry points instrumented to conditionally update a bit in a coverage hash map if not executing in an interrupt handler
    - Novel coverage instrumentation that is sensitive to self-modifying code

Fuzzing VxWorks: Our incredible journey (11/11)

- **<u>Goal</u>: Run target as many times as possible**
- **Challenge: Detecting when the target is "done"**
  - OS kernels (i.e. VxWorks) don't halt unless instructed, so the VM will continue going even if the target is logically "done" processing input
  - No "`idle` function" in VxWorks PPC32
- **Solution: Detect when the kernel goes idle**
  - Summarize execution paths between task schedulings
  - Repeated executions of same code paths signals idleness

DEMO

The path from prototype to production (1/4)

<u>Goal</u>: Speed up the fuzzer to do more executions per second

- **Preserve QEMU code translations between execute-snapshot reload cycles**
  - The VxWorks kernel and target is loaded at the same code locations in every run, so QEMU should not re-translate (part of virtualization) the target machine code that it can take from a prior run
- **Ahead-of-time translation and optimization of target machine code to QEMU TCG**

The path from prototype to production (2/4)

<u>Goal</u>: Make it easier to adopt a new embedded system

- **Key roadblock is lack of emulation support for hardware and devices needed by target software**
  - Fundamental "modelling" issue
  - Symbolic execution may be appropriate (e.g. via the QEMU-based S2E)
  - We anticipate 1 month of effort to bring up a new system, with decreasing integration effort over time as synergies are recognized
- **Need more tooling to help users identify and handle or stub out code that initializes or interacts with devices**

The path from prototype to production (3/4)

<u>Goal</u>: Handle new and unique input sources

- **Currently hook functional interfaces, e.g. POSIX-like I/O, that wrap around devices**
    - Need to implement passthrough for memory-mapped I/O
    - First problem is to even know that direct memory accesses ought to be backed by memory-mapped I/O is not always obvious
- **Targets with rigid interrupt timing requirements, or where the I/O is the sequence of incoming interrupts**

The path from prototype to production (4/4)

Goal: Keeping the tooling up-to-date

- **Depend on two open-source tools (AFL and QEMU)**
  - New Major QEMU release since project started
- **99% of code isolated to PowerFL-specific directories, making upgrading QEMU straightforward**
- **AFL is rarely updated, but keeping up-to-date should not be too challenging**
  - Fun fact: we found a bug in AFL and fixing it makes our fuzzer more effective, so perhaps we are already "ahead"

In summary, we conclude

- **Developed a VxWorks fuzzing prototype for embedded systems**
  - Fuzz a hardware platform without the platform or explosions
  - Doesn’t require the hardware, though hardware knowledge helps
  - Approach is broadly applicable (e.g. to automotive and SCADA systems)
- **Next step is to evolve a production quality capability**
  - Speed up bug-finding capability
  - Speed up adoption time of new targets

Peter Goodman: peter@trailofbits.com
Artem Dinaburg: artem@trailofbits.com
Trent Brunson: trent.brunson@trailofbits.com

Website: https://trailofbits.com
Blog: https://blog.trailofbits.com
Twitter: @trailofbits