ENGR-E 399/599: Embedded systems reverse engineering

Lecture 13: Advanced analysis techniques

Austin Roach
ahroach@iu.edu

April 14, 2022

Today's plan

- Fuzzing
- Symbolic execution
- Emulation

Fuzzing

Automated process for:

- Generating test inputs
- Monitoring the effect of the test inputs on a target binary/system
- Triaging crash-producing or otherwise special test inputs for more detailed analysis

Huge impact on automated testing of applications, libraries, and operating systems

Fuzzing origins

- First fuzzer used to search for bugs in Unix utilities (1990)
- Input generation: random
- Crash detection: check for core files produced by program crashes
- Bugs were found

Generating test inputs: random fuzzers

Random fuzzers:

- Create purely random inputs
- Very easy to implement
- Will likely find only very shallow bugs
  - Low probability of passing even basic input validation tests

Generating test inputs: generational fuzzers

Generational fuzzers:

- Start with some specification of a valid input
  - Model, grammar, protocol description, etc.
- Make random changes both within the specification, and outside the specification
- Increased crash-finding efficiency compared to random fuzzers
- But some difficulty (perhaps substantial) in describing the input specification
- Example: Peach Fuzzer, now GitLab Protocol Fuzzer (https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce) and boofuzz (https://github.com/jtpereyda/boofuzz)

Generating test inputs: mutational fuzzers

Mutational fuzzers:

- Start with an example of valid or special input(s)
- Apply random mutations to the valid or special input(s)
- Removes difficulty of writing a specification
- Possibility for lower quality inputs
- Example: AFL (http://lcamtuf.coredump.cx/afl/) and libfuzzer (https://llvm.org/docs/LibFuzzer.html) and oss-fuzz (https://github.com/google/oss-fuzz)

Mutational fuzzing has greatly increased adoption of software fuzzing

Monitoring

White-box fuzzers:

- Detailed program analysis for path monitoring, constraint solving

Gray-box fuzzers:

- Relatively light weight program analysis and instrumentation

Black-box fuzzers:

- Monitor only external program behavior

Coverage guidance

- Track basic block coverage/edge coverage
- Used to identify inputs that allow access to new regions of program execution
- Requires binary modification, or compilation with a specific compiler to insert instrumentation
- Very powerful technique to drive generation of inputs
- Example: AFL (`http://lcamtuf.coredump.cx/afl/`) and libfuzzer (`https://llvm.org/docs/LibFuzzer.html`) and oss-fuzz (`https://github.com/google/oss-fuzz`)

Fuzzing for reverse-engineering

Goals:

- Bug-finding (security vulnerabilities)
- Identify undocumented commands/protocols

Fuzzing techniques for reverse engineering:

- Blackbox fuzzers
- Blackbox-extensions to graybox fuzzers
  - AFL-QEMU
- Couple fuzzing with symbolic execution techniques (concolic execution)
  - MAYHEM, DRILLER, all the CGC performers...
- Super fancy binary surgery techniques

Fuzzing embedded systems: providing inputs

- Provide over some electrical interface
  - Limited ability to parallelize fuzzing
  - Bandwidth limitations
- Supplied through emulation
  - May be faster
  - Setting up emulation environments can be hard

Fuzzing embedded systems: Monitoring

Need some way to observe crashes, misbehaviors, or system states changes. Options:

- Monitor embedded system directly:
  - Hangs
  - Known responses to control stimuli
- Observe through emulation layer

Detecting misbehavior in some embedded systems can be really hard!

- No segfault for out-of-bounds memory access
  - Just weird behavior 2 hours later...

Fuzzing embedded systems: Coverage guidance

- Debug ports could provide instruction trace information
  - Single-step or otherwise dynamically manage breakpoints in the absence of a trace unit
- Not always easy to set this up
- Emulated firmware gives additional hooks

Embedded system fuzzing summary

- Fuzzing of embedded systems typically requires customized testing environments
- Not as easy as running afl-gcc; afl-fuzz

Fuzzing references

- B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of Unix utilities", *Comm. ACM* 33, 12 (1990), pp.32-44. ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf
- S. Nagy and M. Hicks, "Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing", IEEE S&P, 2019. https://arxiv.org/abs/1812.11875
- M. Zalewski, American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/
- M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: challenges in fuzzing embedded devices", NDSS 2018. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-4_Muench_paper.pdf
- A. Q. Nguyen and K. J. Lau, "Digging deep: finding 0days in embedded systems with code coverage guided fuzzing", Hack in the Box 2018. https://conference.hitb.org/hitbsecconf2018pek/materials/D2T1%20-%20Finding%200days%20in%20Embedded%20Systems%20with%20Code%20Coverage%20Guided%20Fuzzing%20-%20Dr%20Quynh%20and%20Kai%20Jern%20Lau.pdf

Symbolic execution

Means of analyzing program behavior:

- Symbolic, rather than concrete, input values
- For each control flow path, produce a set of constraints on symbolic values
- Satisfiability solver to evaluate statements about execution path
  - Reachability, conditions for producing certain effects, etc.

Symbolic execution is used for formally proving program correctness, and as a tool used for reverse engieering and security evaluations.

Example for a simple program

https://arxiv.org/pdf/1610.00502.pdf#page=2

Symbolic execution summary

Benefits

- Large amounts of semantic insight
- Can evaluate requirements on variables to produce certain results

Limitations

- Potentially demanding computing resource requirements
- State explosion
  - Number of states increases exponentially with branch points
- No natural way to interact with the environment
  - Parameterization or modeling desired for performance

The environment problem for embedded systems

- In symbolic execution of software applications, standard library/system calls are often summarized to simplify the analysis
  - Simple model of functionality to minimize state explosion problem
- For deeply embedded applications, no well defined standard library/system call interface
- Need some way to handle peripheral interaction in embedded systems

Firmalice

Goal:

- Identify authentication bypass vulnerabilities in firmware
  - Access to firmware or information disclosed by system gives information to reach privileged program points

Inputs:

- Policy describing what authentication bypass looks like
- "Symbolic summaries"–test cases for standard library function ID and symbolic descriptions of operation

Approach:

- Automated identification of base address by heurestic analysis of jump tables
- Automated identification of entry points as weakly connected nodes in control flow graph
- Backward slicing from privileged program points
- Forward symbolic execution of identified slices

Firmalice outcomes

Trickery:

- I/O identified by a set of heuristics
  - Network connections in program slice of user-space firmware assumed to represent user input
  - For bare metal firmware, values coming from interrupts that concretize primarily to ASCII values assumed to be user input
  - Values used by interrupts that concretize priarily to ASCII values assumed to be user output
- Lazy analysis of initialization procedures
  - Keep track of references to memory regions
  - If uninitialized value needed to evaluate program state, execute appropriate initialization routine

Results:

- Identified authentication bypasses in several consumer products
  - Hard-coded credential in HTTP authentication for networked camera
  - Hard-coded SNMP community string allowing access to networked printer

FIE on firmware

Goal:

- Identify memory handling vulnerabilities in firmware for MSP430 microcontroller

Inputs:

- Source code for multiple open-source MSP430 firmware repositories

Approach:

- Customization of KLEE to support 16-bit microcontroller
- Symbolic execution, assuming unconstrained memory reads (from memory mapped I/O) can take on any value
- Allow interrupts after any instruction
- State pruning and memory smudging to avoid state explosion

FIE on firmware outcomes

Trickery:

- Identification of ‘out-of-bounds’ memory accesses reliant on type/structure information provided by source code
- Possibility that identified errors not possible when real peripherals connected to I/O

Results:

- Found some bugs
- Demonstrated memory safety for some projects

FirmUSB

Goal:

- Identify malicious behavior in firmware of USB peripherals

Inputs:

- Binary firmware images for devices using 8051/52 microcontroller architectures

Approach:

- Developed custom lifters for 8051 to VEX and LVM IRs
- Symbolic execution using Angr and FIE (KLEE)

FirmUSB outcomes

Evaluating claimed identity:

- Scan binary for constants used in device descriptors
- Identify potential output buffer locations through data flow analysis
- Look for incorrect device descriptors being written to output buffer

Evaluating correctness of operation:

- Look for concrete values written to output
  - Suggests hard-coded data injection rather than relay from user input/environment

Results:

- Evaluation approach is somewhat circular...
- But identification of malicious functionality of two doctored firmware samples

Avatar

Goal:

- Address environment problem by interacting with system hardware for peripheral interaction

Inputs:

- System firmware and debug interfaces to hardware

Approach:

- S2E symbolic execution engine
- Multiple examples of debug interfaces
  - JTAG debug access ports
  - Customized debug firmware
- Context switching to change between symbolic execution and monitored execution on hardware
- Upload/download system state during context switches

Weaknesses:

- Context switching/monitored execution are very slow
  - Big problem for systems with timing constraints

Results:

- Successful analysis of several diverse embedded systems

Inception

- Similar approach to Avatar
  - Consult hardware for peripheral interaction
- Combine source code (high amounts of semantic information) with hand-coded assembly lifted to IR (lower amounts of semantic information)
- Custom-built JTAG controller for speedy access to DAP

Symbolic execution references

- R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques", *ACM Computing Surveys*, 51, 3, 2018. https://arxiv.org/abs/1610.00502
- E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask", *IEEE S&P*, 2010. https://users.ece.cmu.edu/~aavgerin/papers/Oakland10.pdf
- Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "(State of) the art of war: offensive techniques in binary analysis", *IEEE S&P*, 2016. https://ieeexplore.ieee.org/abstract/document/7546500
- Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware", *NDSS*, 2015. https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/firmalice-automatic-detection-authentication-bypass-vulnerabilities-binar

Symbolic execution references continued

- D. Davidson, B. Moench, S. Jha, and T. Ristenpart, "FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution", *22nd USENIX Security Symposium*, 2013. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson
- G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. B. Butler, "FirmUSB: vetting USB device firmware using domain informed symbolic execution", *Proc. ACM SIGSAC Conf. Comp. and Comm. Sec.*, 2017. https://arxiv.org/pdf/1708.09114.pdf
- J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: a framework to support dynamic security analysis of embedded systems' firmwares", *NDSS* 2014. https://www.ndss-symposium.org/ndss2014/programme/avatar-framework-support-dynamic-security-analysis-embedded-systems-firmw
- N. Corteggiani, G. Camurati, and A. Francillon, "Inception: system-wide security testing of real-world embedded systems software", *27th USENIX Security Symposium*, 2018. https://www.usenix.org/node/217621

Emulation

- Supports dynamic analysis when no debug access is available
- Allows parallelization of analyses that might otherwise be limited by the number of physical systems

PANDA

Goal:

- Whole-system emulation framework for reverse engineering

Inputs:

- System firmware images

Approach:

- Built on QEMU
  - Supports a large set of instruction set architectures
- Ability to record and replay executions
  - Efficient storage for long instruction traces

Approach (cont):

- Plug-in architecture for different analyses
- Architecture-neutral analysis

Weaknesses:

- Performance sometimes slow
- Record/replay not supported by all architectures

Results:

- They analyzed some firmware!

P2IM

Goal:

- Emulate system firmware to allow fuzzing of emulated peripheral interfaces

Inputs:

- System firmware images

Approach:

- Modified version of QEMU
  - Allows control of memory accesses
- Identify type of memory-mapped peripheral accesses by access patterns
  - Control registers, data registers
- Supply random data for data register accesses

Weaknesses:

- Random data may limit firmware execution
  - Might find only shallow bugs

Results:

- They found some bugs!

Avatar2

Goal:

- Co-ordinate multiple dynamic binary analysis tools to efficiently analyze execution

Inputs:

- System firmware images
- Debugging interfaces
- Emulator tools

Approach:

- Multiple tools
  - GDB
  - OpenOCD
  - QEMU
  - PANDA (QEMU-based reverse engineering tool)
  - angr (for symbolic execution)

Approach (cont):

- Move execution of firmware from one tool to another
- Use physical hardware for input/ouput and memory accesses to peripherals

Weaknesses:

- Transfer of execution is inefficient
- No generic support for interrupts

Results:

- They executed and analyzed some firmware!

HALucinator

Goal:

- Allow re-hosting and emulation of firmware by replacing hardware abstraction layer (HAL) functions

Inputs:

- System firmware images
- Identification of HAL functions

Approach:

- Replace HAL functions with user-provided functionality
- API for handling inputs/outputs from peripherals
  - Monitor traffic
  - Provide pre-determined inputs
  - Fuzz inputs

Weaknesses:

- HAL functions not always easily identifiable
  - No symbols
  - Use of unusual HALs

Results:

- They executed and analyzed some firmware!

Emulation references

- B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable Reverse Engineering with PANDA", *PPREW-5*, 2015. https://apps.dtic.mil/sti/pdfs/AD1034415.pdf https://panda.re/
- B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-indepedent firmware testing via automatic peripheral interface modeling", *USENIX Security*, 2020. https://www.usenix.org/conference/usenixsecurity20/presentation/feng https://github.com/RiS3-Lab/p2im
- M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar^2: a multi-target orchestration platform", *BAR* 2018. https://www.eurecom.fr/en/publication/5437 https://github.com/avatartwo/avatar2
- A.A. Clements et al., "HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation", *USENIX Security*, 2020. https://www.usenix.org/conference/usenixsecurity20/presentation/clements