MacOS/iOS Kernel User fuzzing training

Description:

This training will focus on macOS kernel and user-mode fuzzing, with an emphasis on kernel fuzzing. Instead of relying on off-the-shelf tools, it will teach you how to develop your own fuzzer from scratch. You'll learn techniques such as coverage-guided fuzzing through binary rewriting, identifying and analyzing kernel targets, and manually building each step of the process. This is a hands-on, low-level approach designed to build a deep understanding of fuzzing internals.

Part 1: iOS/macOS Kernel Fuzzing

Enumerating Attack Vectors in Kernel Fuzzing

- System calls (Mach subsystem and Unix subsystem)
- IOKit interfaces and services (`IOServiceOpen`, `IOConnectCallMethod`, `IOUserClient`, ...)
- Review of sandbox and accessible attack vectors (entitlements and MAC **sandbox policy**)
  - Case study: **BlastDoor** and other sandbox profiles
  - Sandboxing an executable
- Enumerating accessible attack surfaces inside a sandboxed context (system calls, IOKit, hidden attack surfaces like file systems, ...)
- Memory vulnerabilities: race conditions, use-after-free, and buffer overflow vulnerabilities.

Introduction to Fuzzing

- What is fuzzing?
- Dummy fuzzing
- Basic bit-flipping kernel fuzzing via **dylib** injection
- Bypassing entitlements
- KCOV, coverage-guided fuzzing: basic concepts (basic block, edge coverage)
- KASAN, Address Sanitizer

Tools for Kernel Fuzzing

- Reversing macOS **kernel extensions** with **Ghidra** (`IOUserClient`,
  - `externalMethod`,
  - `getTargetAndMethodForIndex`)
- Basic kernel functions tracing with DTrace.
- Binary rewriting and kernel collection:

    - Introduction to **Pishi**, a tool developed by the instructor to fuzz closed-source KEXTs as if they were open-source.
- **Pishi KCOV**: instrumenting the kernel or KEXTs for coverage
- **Pishi KSAN**: kernel Address Sanitizer for KEXTs
    - Ghidra setup and environment
    - Ghidra scripting
    - Pishi kernel architecture (trampoline, shared memory, ...)
    - Kernel binary rewriting in Ghidra
    - Collecting coverage
    - Binary address sanitizer
- **Fuzzing Environment**
    - How to load a custom-instrumented kernel and boot macOS using **Virtualization.framework** and **Parallels Desktop**
    - macOS Kernel debugging with **Virtualization.framework**

Writing Fuzzing Harnesses for Kernel Targets

- **Stateless** or binary fuzzing
- IOKit fuzzer using LibFuzzer/libAFL
- Feeding kernel code coverage feedback to LibFuzzer/libAFL
- Using **Pishi** and **Ghidra** to tune the fuzzer (e.g., spotting target functions in Ghidra and performing targeted instrumentation)

Structured-Aware and Stateful Fuzzing

- Fuzzing with structured data
- Fuzzing system calls using **libprotobuf-mutator** and **Pishi**

Triaging a Kernel Panic and Finding Vulnerabilities

Part 2: iOS/macOS User Mode Fuzzing

- Fuzzing in user mode
- Fuzzing open-source libraries
- Fuzzing closed-source binaries
- Introduction to the **Jacklope** fuzzer and comparison with other macOS binary fuzzers
- Implementing harnesses for various frameworks and libraries:
    - Image
    - Font
    - Video

    - Audio
    - ...
- Optimizing the harness and enabling guard pages
    - dyld shared cache
    - TinyInst Hooks
- Grammar-based fuzzing
- macOS IPC mechanisms and fuzzing for sandbox escapes:
    - XPC
    - Mach IPC
- Identifying attack vectors for sandbox escapes
- Fuzzing iOS dylibs on macOS
- Fuzzing and fun with **Frida**

Training Requirements

Student Requirements

- Basic understanding of exploitation
- Knowledge of C and Python programming
- Ability to understand simple ARM64 assembly

Hardware Requirements

- MacBook with Apple Silicon (M series)
    - Root access — disabled SIP

Software Requirements

- Xcode and Ghidra, LLDB, CMake installed (recommended)
- Additional software will be provided during the training session