Discovering Vulnerabilities in Embedded Firmware with Fuzzing Techniques

D. Vizár, M. I. Ben Salah, M. Marty

Connected embedded devices are exposed to adversarial interactions, making exploitable firmware bugs a major concern for their security. CSEM developed an experimental embedded firmware test bench, combining fuzzing and hardware emulation, to identify and eliminate the maximum of the firmware bugs before the firmware is deployed.

The presence of programming errors (a.k.a. bugs) that constitute an exploitable vulnerability, such as the well-known buffer overflow vulnerabilities, are among the dominant attack vectors used by cybercriminals [1]. As applications of embedded devices evolve to incorporate various forms of untrusted input, such as inputs received over the user interface, firmware update packages and generally any received communication frames, the issue of exploitable bugs becomes a pressing one in this context as well. For example, an exploitable buffer overflow in the BLE communication stack would allow attackers in the vicinity of a device to attack it remotely, without the need of physical interaction. By exploiting such a vulnerability, an attacker might execute arbitrary code on the targeted device. The impact of such an attack can be devastating on constrained devices, as they typically lack features such as a full-fledged memory protection unit able to contain the vulnerability. Even though recent devices do feature technology such as ARM TrustZone, these only separate the most sensitive resources from the rest of the system and do not fully isolate processes, for example. To make things worse, exploitable vulnerabilities are frequently imported by 3rd party software modules.

Fuzzing is one of the techniques able to reduce the number of exploitable bugs in software. It consists in feeding a program under investigation with large amounts of varying inputs, until an input is found that forces the program to exhibit an unexpected, or faulty behavior. The software managing this process and choosing how to vary the test inputs is called a fuzzer. Modern fuzzers are designed to optimize the mutation of test inputs such that an error-provoking one is found faster with various techniques, either by observing the black box behavior of the program binary (black box fuzzing) or by working directly with the source code (white box fuzzing). Fuzzers are indeed effective tools for identification and elimination of bugs in software, however, because they need to fully control and manipulate the execution of software target, they are intractable on embedded systems, which do not have enough memory and computational power to run both the firmware and the fuzzer.

Recent academic research proposed new approaches and tools that enable the fuzzing of embedded firmware. One such tool called HAL-fuzz [2] uses an emulation-based approach (see Figure 1). The target firmware binary is executed on the host, in a CPU emulator, such that arbitrary symbols in the binary may be intercepted and their execution skipped, modified, or mocked. This allows drivers and peripherals to be efficiently mocked in a way that is consistent across several embedded platforms; when a driver function is called, the emulation is stopped, the effects of the driver function call evaluated in high-level language and the emulated system is modified accordingly, jumping over the driver call. The same approach also allows for the injection of fuzzing inputs as well as assertions on the system state to be checked. Because the framework work with a binary but requires symbols to be intercepted, it represents a gray-box approach.

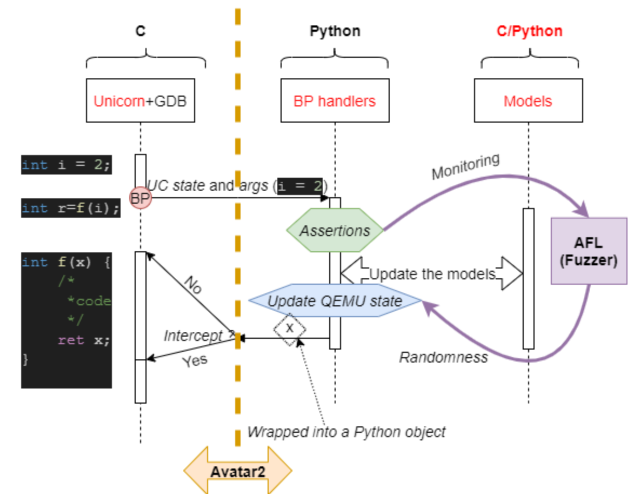

Figure 1: Embedded firmware fuzzing based on the HAL-fuzz framework, combining Unicorn CPU emulator, Avatar2 emulation control framework and AFL Fuzzer.

At CSEM, a proof of concept of an embedded fuzzing testbench has been implemented using the HAL-fuzz framework, with the ultimate goal of integrating an automated fuzzing step into the continuous development pipeline of critical firmware modules. In the proof of concept, the CSEM proprietary ultra-low power real-time operating system µ111 has been designated as target and a mock console utility has been implemented with a purposefully included buffer overflow bug. The HAL-fuzz framework has been deployed, emulating the execution on ARM Cortex M3. Handlers have been implemented to intercept all input/output driver calls of a UART console, which has been emulated in Python. The detection of buffer overflow has been implemented with help of *canaries*. In this configuration, the test bench was able to successfully identify the buffer overflow vulnerability, with a total runtime of ~30 min and 0,03 fuzzer executions per second.

This result confirms that complex embedded firmware, such as a real-time operating system, can be effectively fuzzed. The setup is also compatible with integration in a continuous development environment, where vulnerability testing of critical software modules (e.g., communication stack) could be run automatically. For that purpose, performance optimizations are expected, e.g., by executing only the targeted software module from a snapshot. It is also necessary to consider the fuzzing of third-party software modules as well as to design generic, target-code-independent assertions for detection of memory leaks.

[1] MITRE "2023 CWE top 25 most dangerous software weaknesses", https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

[2] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, M. Payer, "HALucinator: firmware re-hosting through abstraction layer emulation", USENIX (2020).