Triforce QNX Syscall Fuzzer

Pallavi Pandey
Computer Science and Engineering
Indian Institute of Technology, Bhilai
Raipur, India
pallavip@iitbhilai.ac.in

Anupam Sarkar
Security Verification and Validation (ESY)
Robert Bosch Engg. & Business Solutions
Bangalore, India
anupam.sarkar@in.bosch.com

Ansuman Banerjee
ACMU
Indian Statistical Institute
Kolkata, India
ansuman@isical.ac.in

***Abstract*—The task of mitigating kernel vulnerabilities in a RTOS kernel like QNX is of utmost importance in recent times. AFL is probably one of the most effective fuzzing tools available, with its functionalities for feedback driven and instrumented fuzzing. In this paper, we present our experience report on developing an environment for fuzzing QNX kernel using AFL.**

***Index Terms*—Kernel fuzzing, American Fuzzy Lop (AFL), QNX, QEMU**

I. Introduction

Fuzzing is a technique used for security testing of software implementations. Fuzzers produce random and even invalid data as inputs and then monitor the program under test for faults, crashes, timeouts etc. Kernel Fuzzing is used to test vulnerabilities in OS kernels. Dumb fuzzing techniques find it difficult to fuzz kernels effectively since the mitigation of kernel vulnerabilities can involve techniques like user-mode privilege separations. We need to provide specific types of inputs to fuzz OS kernels, specifically syscall inputs that trigger system calls. System calls, which are generated in user space, when invoked, transfer the control to the OS kernel. The main philosophy behind our proposal is to create a wrapper around system calls and model them in a suitable way, to pass them as inputs to the fuzzer. If a crash or vulnerability is detected, it is stored in form of crash / hang reports in the user space. Normal dumb fuzzing like QNX syscall fuzzer [3], uses random or even syscall inputs that would not work effectively. The main advantage of using state-of-the-art fuzzers like AFL will be its feedback mechanism, which empowers it to generate "intelligent" inputs. In the following, we discuss our overall experience in using this technique for fuzzing the QNX kernel.

II. Methodology

In this section, we first present an overview of TriforceAFL, on top of which our framework was built. This is followed by an overview of the working principle of our method.

A. TriforceAFL

For fuzzing the QNX kernel, we extended the TriforceAFL [1] framework. While the conventional AFL could only fuzz programs or binaries built using afl-clang or afl-gcc, the support of QEMU user-mode emulation allowed to fuzz even random binaries running in the QEMU emulator. TriforceAFL is a modification of the conventional AFL, which supports QEMU's full system emulation. Hence, using this tool, our AFL target can be an OS kernel, and we can fuzz the VM running under QEMU's full system emulation. We present below a brief overview of the working of TriforceAFL to fuzz any OS kernel.
The Host OS which has TriforceAFL built on it, runs QEMU and AFL. The VM running on QEMU is the target. When this guest VM boots on QEMU, it runs a *driver* program. This driver program communicates with AFL, QEMU and also the Host OS, and is the main crux of the kernel fuzzing step using TriforceAFL. Various other communications also occur, where QEMU, AFL and Host use Unix pipes to communicate with each other. The driver communicates with AFL, QEMU and also the Host OS using hypercalls. The overall architecture of the framework is shown in Figure 1.

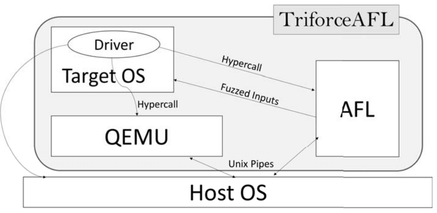

Fig. 1. Architecture of TriforceAFL

We now describe the overall working principle. TriforceAFL boots the OS in QEMU's full-system emulation mode. The driver program, which starts on the guest-VM startup, makes the aflCall *startForkserver*. This starts the AFL's forkserver, inside which the usual AFL procedures are followed. This forkserver creates a fork of the VM for running each test case. Once this is done, everything executes inside this forkserver. The driver, which runs in a fork of VM makes a hypercall to startForkserver() to start the AFL's forkserver for other test cases. The driver program also makes a hypercall to getWork() to put the serialized test cases into the buffer and then de-serializes the test case. It then makes a hypercall to startWork() to AFL to start edge-tracing. The driver is responsible for execution of the test cases and then reports the test crash or makes a hypercall to endWork() if it does not crash.

B. High level overview of the method

In this section, we provide a description on how to fuzz a QNX kernel, where its target VM runs under QEMU's full-system emulation. At a high level, the most important challenge is to get the driver program to run on QNX emulated on QEMU, which can communicate with AFL. Our reference for this driver program was highly influenced by the *TriforceLinuxSyscallFuzzer* [2], a tool used for fuzzing Linux kernels. TriforceLinuxSyscallFuzzer takes an init ram disk and a kernel image bzImage (which in case of Linux is vmlinuz). The approach needs to create a root template which acts as the init root directory and then to include the driver program in its startup. We first need to boot QNX on QEMU. On the target VM (QNX) startup, the driver program also needs to be started. Further, we also need to model QNX syscall inputs so that they can be fed as seed inputs to AFL. In the following section, we present a discussion on our experiments.

III. Experiments

For fuzzing the QNX kernel using TriforceAFL, we worked with the TriforceQNXSyscallFuzzer — which contains all driver files, bootable images and other scripts and files to build an environment compatible with TriforceAFL. The first task was to emulate QNX on the QEMU version supported by TriforceAFL. When we build TriforceAFL, it generates a binary file called *afl-qemu-system-trace*, which is the binary for QEMU full system emulation for the x86_64 architecture. *afl-qemu-system-trace* is an exact replica of the qemu-system-x86_64 command. For executing QNX on QEMU's full-system emulation, we had two files, namely, fs.vmdk and ifs.vmdk (QNX filesystem and image-filesystem files), which are obtained from the stripped-down version of QNX emulated on VMware. IFS consists of startup program, OS Kernel, build scripts and other device drivers. FS, on the other hand, is the filesystem to execute outside the kernel. In the TriforceLinuxSyscallFuzzer, the task to boot the VM was by using the initial ram disk and kernel image using the following command line.

```
$ ./afl-fuzz -t 500+ -i inputs -o outputs -QQ -- \
    ./afl-qemu-system-trace \
   -L ../TriforceAFL/qemu_mode/qemu/pc-bios \
   -kernel bzImage -initrd ./fuzzRoot.cpio.gz \
   -m 64M -nographic -append "console=ttyS0" \
   -aflPanicAddr "$PANIC" \
   -aflDmesgAddr "$LOGSTORE" \
   -aflFile @@
```

For QNX, it is enough to use fs and ifs for QEMU's full-system emulation. However, fs and ifs need to be modified so that the driver program runs when QNX boots on QEMU. We built the driver executable by compiling and linking its C files on QNX Momentics IDE. Then we transferred this executable to QNX running on VMware. Now, when the QNX boots, the initial script which executes is 'startup.sh' located in the 'etc/' directory of the root directory of QNX. We modified this script such that it immediately starts the driver after configuring the OS. We extracted the modified fs and ifs, which, when used, can immediately trigger the driver program and thus, communicate with TriforceAFL. FS and IFS were passed to the '-drive' option of *afl-qemu-system-trace*, as shown in the code snippet for QNX-kernel fuzzing.

Now, to begin the actual fuzzing with TriforceAFL, we used the binary *afl-fuzz* in TriforceAFL. We also needed some seed inputs to provide to AFL. Currently, for this purpose, we used the POSIX system calls which were initially modelled for fuzzing Linux kernels. Being of POSIX standards, they also act as good seed inputs for fuzzing QNX kernels. The crash / hang reports and the outputs of fuzzing are stored in the 'outputs' directory and can be viewed in real-time, while the fuzzing is on-going. The command line for the above task is given below.

```
1   $ ./afl-fuzz -t 50000+ -i inputs -o outputs -QQ -- \
2         ./afl-qemu-system-trace \
3        -m 256M \
4        -L /TriforceAFL/qemu_mode/qemu/pc-bios \
5        -drive file=ifs.img,format=raw \
6        -drive file=fs.img,format=raw \
7        -aflPanicAddr $PANIC \
8        -aflDmesgAddr $LOGSTORE \
9        -aflFile @@
```

The *afl-fuzz* takes an input directory (with -i option), which contains the seed inputs and an outputs directory (specified with -o option), for storing the reports / results of fuzzing. The -t option is used to set the timeout limit to 50000 milliseconds and above. It was observed that in case of QNX kernel fuzzing, that test cases resulted in an *infinite loop and subsequent timeout* and were skipped with warnings, if a lower timeout limit was specified. Then, there is the -QQ option, which is the extended functionality of TriforceAFL to support QEMU's full-system emulation. Lines 2-6 specify the target binary, which is the QEMU binary afl-qemu-system-trace with its full command line for emulating QNX on QEMU. '-aflPanicAddr' and '-aflDmesgAddr' are used for specifying the required panic and log_store addresses. The QNX kernel fuzzing process can be depicted as follows:

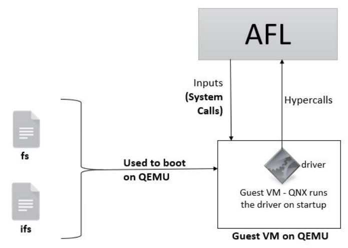

Fig. 2. TriforceQNXSyscallFuzzer: a high-level perspective

IV. Conclusion

We got the TriforceQNXSyscallFuzzer up and running, which is efficient from the increase in levels and coverage observed on AFL status screen. We are constantly modelling on making the kernel fuzzer more context aware to fuzz effectively.

References

[1] T. Newsham, J. Hertz, 'TriforceAFL', 2016. [Online]. Available: https://github.com/nccgroup/TriforceAFL. [Accessed: 6- Jul- 2019].

[2] T. Newsham, 'TriforceLinuxSyscallFuzzer', 2017. [Online]. Available: https://github.com/nccgroup/TriforceLinuxSyscallFuzzer. [Accessed: 12- Jul- 2019].

[3] A. Plaskett, 'QNX Security Tools', 2016. [Online]. Available: https://github.com/alexplaskett/QNXSecurity. [Accessed: 11- Jul- 2019].