Fuzzing, an exploration of Mind, Body, and Argentina

Stephens, Nick
stenic05@evergreen.edu

Boesen, Stefan
stefan.boesen@gmail.com

David, Max
davmax04@evergreen.edu

June 29, 2014

Abstract

This paper intends to summarize a quarter of undergraduate research in *fuzzing*. In Section 1 we introduce fuzzing, methodologies, and best practices. Section 2 describes our experience and tactics implementing those techniques on a real world library, the *Music Player Daemon* project. Section 3 contains our results, closing thoughts, and words of advice for those interested.

1 Introduction and Intent

Our learning contract was intended to expand our understanding of fuzzing and how it applies to real world projects. Fuzzing: Brute Force Vulnerability Discovery [cite] was our primary reference, being a well known introductory book on the topic. Conceptually fuzzing is a very simple topic. By feeding unanticipated input into applications that do not properly handle them, unexpected behavior can be introduced into otherwise functioning programs. As an example, a program may attempt to read a number from a network connection. What happens if the program is sent a character instead? Is the character interpreted as a number, or does it just fail?

The goal of fuzzing is typically to identify locations of potential bugs. Typically this entails having a component reading the "state" of some program, whether that's the error log, signals sent, or other behaviors. As the goal is to produce unexpected behavior, it can be difficult to know if you have anticipated all of the possible states. Fuzzing has other challenges as well, such as identifying inputs to target as well as determining what input to generate. Instead of testing every number, perhaps it's more valuable to try extremes and random ones throughout. Strings with unusual characters may be preferred.

A final challenge is that simplistic fuzzing measures the state after each input, when in reality, a system's state is affected by previous commands as well. This means that an environment should be set up where one can return to a previous state quickly (using a local virtual machine with memory snapshotting) can be critical for testing sequences of inputs.

Our goal going into this project was to implement a basic fuzzer in order to discover vulnerabilities in an open source program we had all used. We chose *Music Player Daemon*, which we'll refer to by its initials, *MPD*.

2 Implementing the Fuzzer

A fuzzer is made up of multiple components. Our fuzzer in particular had two. As mentioned previously there is more to fuzzing than inputing provocative data to the target, one must also have some method of monitoring the target's condition. To monitor the state of MPD as we sent fuzzer data to it, we leveraged Linux's debugging suite *ptrace*.

When *ptrace* is attached to a target process, it allows the tracer to intercept IPC signals sent to the target. These signals are often indicative of a program crash and can tell us if the program attempted to access a non-existent memory region or execute an illegal instruction, among other things. With *ptrace* we can then dump the callstack and register state for further analysis as well as alert our fuzzer, if we wish it to stop.

MPD also has many input channels to consider. Possible attack surfaces could range from a malicious *Ogg* file constructed to exploit a vulnerability in the file parser, to the remote command interface MPD provides. Our fuzzer only targeted the latter. Developing the fuzzer incrementally, we began by sending completely random strings of data to a *ptrace* monitored MPD server. Next we sent commands followed with arguments composed of random bytes. Finally we began sending arguments which were type correct, but random within the type domain, commands would also be sent in random order, in an attempt to mutate the server's internal state, and gain code coverage.

3 Conclusion

Although not as widely deployed as some other big name projects, *Music Player Daemon* is a mature open-source project that follows a thoughtful development process with a sizable collection of developers. Patches are vetted through the Clang Static Analyzer to find any common programming errors. Although we could find no history of security audits in the past, an active mailing list shows that new additions are continually reviewed. We were very impressed with the level of organization within the project codebase.

Through our process of fuzzing MPD, specifically its remote command interface, we did not come across any unintended or otherwise bizarre reactions. The command module responded robustly to our barrage of nonstandard inputs, as

described in section 2. With more time spent on the project, we believe fuzzing the file parsing capabilities may prove more fruitful. MPD deals with a large number of varying file formats, that historically has held many bugs in similar software.

Naturally, our inability to find any bugs does not preclude their existence. Any penetration test of this sort could never prove the security of a software. Fuzzing in particular is a rather "dumb" way of finding vulnerabilities; we're throwing random inputs at a program to see what happens. What our fuzzing does accomplish is improve our confidence in MPD's security by assuring a much smaller probability of low-hanging fruits in command parsing. Indeed many bugs have been found from a quick fuzzing exercise, and is usually an excellent first step in any bug hunting endeavor.