OSS Reproducer*

1 Introduction

Opensource software is a significant part of people's life. Also, it's a great source to perform security research as well. In this report, we built a scalable database of vulnerable open-source Software based on google's OSS Fuzz project by implementing an OSS Fuzz reproducer. With the ability to reproduce arbitrary bugs and the open-source software version control history, we can locate the vulnerable codes and corresponding fixes for each vulnerability. Furthermore, with the ability to build vulnerable software from source code, we build a framework to evaluate the vulnerability fixer. Combining our real-world vulnerability database, we can evaluate different strategies of a specific vulnerability fixer or kinds of vulnerability fixer.

2 Motivation

Building the Vulnerability Database Due to the lack of a scalable source of vulnerabilities, we find the size of the testing database is limited in related research and the vulnerabilities are not fresh(there are years of lag). As the new development framework generates, currently running software is less likely to use a library that was popular 10 years ago. Therefore, using a fresh database can better process your security research, such as testing the vulnerability discovery ability, analyzing the vulnerability distribution, and evaluating your vulnerability auto-fix strategies. Our database is based on google's OSS Fuzz project, which collects the freshest vulnerabilities and has been keeping increasing for years. With this great source of fresh vulnerabilities, we can build a scalable real-world vulnerability database. Since the instability of reproducing, we also archive the successful cases. For people who focus on binary(the re-compiling is still not stable), they can get the crash input, vulnerable and fixed binary file by downloading our docker images. More details to build the scalable, fresh, real-world vulnerability database would be included in Section 3.

Collecting the Vulnerable Source Codes After having the ability to reproduce vulnerabilities, we found another very helpful resource available: vulnerable codes. As it can be a valuable resource to train your auto-vulnerability-fixer module. Or you can extract some features from this vulnerability and develop some methods to find similar vulnerabilities. This feature is easy to achieve for OSS projects since we can find all modifications through the version control tools. With the POCs of crashes, the binary search can help us to locate the code and fix the vulnerabilities. We'll introduce our methods in Section 4.

Evaluating Vulnerability Fixers As new technologies develop, Artificial Intelligence and Deep Learning are used to help people fix the vulnerabilities. Based on the advantage of compiling from source code, we can use our database to evaluate different vulnerability auto-fixers and different strategies. Since all the cases come from the real world, it can better show us the complexity of fixing a vulnerability as a human. Also, it provides a great resource to improve your vulnerability fixers. We built a framework and provide APIs to test your fixers and strategies. By accessing the description of vulnerabilities and the feedback of testing results, people are able to adjust their modules or strategies to achieve

*A Report for CE-GY 9963

better performance in the real world. We'll introduce this part in Section 5.

3 OSS Reproducer

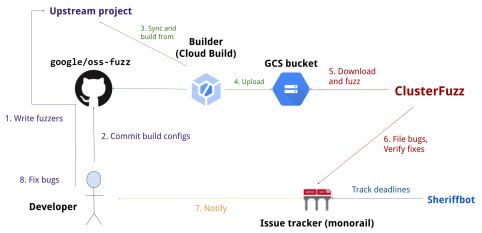

Figure 1: OSS Fuzz Structure

OSS Fuzz OSS Fuzz is one of Google's Open Souce Projects. According to figure 1[1], the project developers write the fuzzers for their projects and commit the build file to oss-fuzz. After the commit is merged, Google runs the fuzzers on clusterFuzz and delivers the vulnerability report to the developers to inform them to fix the vulnerabilities. After the fixes are confirmed, Google publics the vulnerability report, such as a report in the figure 2[2].

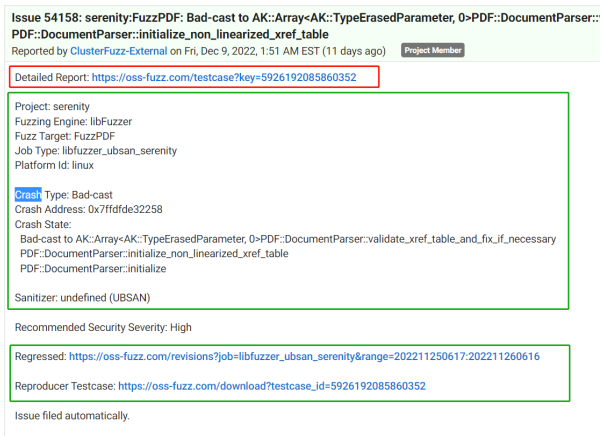

Figure 2: OSS Fuzz Report

In figure 2, the URL in the red box is only available for project developers. As security researchers, we can't achieve detailed reports. However, there is still some valuable information in the report: project and crash information. including the fuzzer target's name, the sanitizer's type, crash type, and address. Also, there is another significant resource we can use in the second green box. The first link shows us the dependencies of the vulnerable software and the second link is the input that results in the crash.

Reproduce the Vulnerability Technically, with the dependencies, we can build the vulnerable fuzz target. But that is not easy. There is an official OSS-Fuzz vulnerability reproducer[3] in Google's repository. However, it doesn't work on plenty of bug reports.

Improvement We'll compare our implementation and Google's reproducer in Section 6. Based on the official one, in order to achieve a higher reproduction rate, we re-write the reproduction procedure. After analyzing Google's reproducer and solving these tricky issues, we implement a better reproducer. Depending on our new reproducer, we built a database with more than one thousand cases. Moreover, with the growth of OSS-Fuzz, our database grows, too. Compared to Google's reproducer, our reproducer mainly has the following advantages.

- Rollback all dependencies to the old version according to the bug reports
- Fix improper dockerfile, build script, and map the URLs of moved dependencies
- Get more precise timestamps to find a better OSS-Fuzz builder container

Rollback Dependencies After reading the source code official reproducer, we found there is no part to get the version of dependencies since Google's developers use the latest version of dependencies because Google's reproducer assumes that the dependencies don't matter and only rolls back the main projects to a certain version. In testing, we found it's highly likely to fail to compile the old version fuzzer target with the latest dependencies. Therefore, we

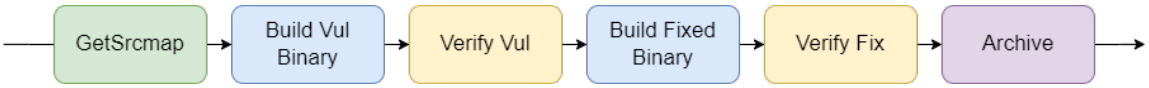

Figure 3: Reproducer Workflow

grab all the version information of dependencies from Google's build bucket. Based on the commit hashes got from the build bucket, we try to check out as many dependencies as possible.

Fix Improper Configurations Also, the old configurations may not be correct. LLVM is a great example of this case. Five years ago(2017), LLVM-Project used SVN as its version control tool and its old links are not available now. We have to deal with these issues because libFuzzer is a part of the LLVM project and there are about 30% cases that use libFuzzer as their fuzzes. We check the dependency names and use corresponding scripts to fix the improper Dockerfiles and build scripts. Because this work is time-consuming and not efficient, we only focus on improper configurations that influence plenty of cases.

Get Correct Base Image Furthermore, we found the OSS Fuzz used a common base image rather than a specific project base image because it accelerates the fuzzing by pre-downloading the popular fuzzer's code to the image. And as we talked about before, vulnerability reproducing is dependency sensitive. So we'd better choose a correct base image to improve the success rate. Google's reproducer only considers the vulnerable commit's timestamp while our reproducer checks all dependencies' timestamps and finds a base image before the earliest timestamp.

Implementation With solving the problems mentioned in the previous 4 paragraphs, We implemented a new reproducer based on the workflow in figure 3. First, we get the source maps of the commit, where the vulnerability is found and fixed. The source maps include the version information of all dependencies so we can download the source code of dependencies and roll back them to the old versions. To make sure we build the binary correctly we should compile and verify the fuzz targets twice: We compile the vulnerable version and confirm the crash input would trigger the crash while the crash input should not crash the fixed version binary.

Limistations Although, we spent much time increasing the success rate of reproducing, there are still a great number of not reproducible cases.

```
...
FROM gcr.io/oss-fuzz-base/base-builder
RUN apt-get update && apt-get install -y make
RUN git clone --depth 1
    https://github.com/openssl/openssl.git
RUN git clone --depth 1 --branch
    OpenSSL_1_1_1-stable
    https://github.com/openssl/openssl.git
    openssl111
RUN git clone --depth 1 --branch openssl-3.0
    https://github.com/openssl/openssl.git
    openssl30
WORKDIR openssl
...
```

Code 1 shows a typical Dockerfile of OSS Fuzz images. For this type of dockerfile, we can remove "git clone" commands and mount the right version dependencies. However, we found there are lots of cases not reproducible because of using curl/wget to get resources from an outdated URL. In this case, the versions of dependencies would not be documented on the build bucket. Several months later, the link expired and there is no efficient way to fix this case and download the right dependencies. Because binary compiling is complex and dependency sensitive, there are lots of issues that may lead to failure. With months of effort, we finally reproduced more than 1279 cases out of 8248 cases.

Scalebility and Reproducibility Since we can use the reproducer to rebuild the binary, the database keeps growing and updating. Furthermore, not like the old and tricky cases, it's easy to reproduce the new vulnerabilities because almost all dependencies are still available. After noticing that a great amount of labeled vulnerable binary is a great resource for security research, we got the idea to archive all vulnerable binary. We pack the compiled binary with its crash input and the source code of its dependency as a docker image so people can pull the image anytime. Even if the compilation might not work a couple of years later, people can still use the binary and its OSS Fuzz report to perform their security research.

4 Fixes Locator

In the previous section, we are able to build the fuzzer target based on the given dependency versions(commit hashes). With this ability, we can move forward and achieve more valuable data: vulnerable code segments and their fixes.

In real life, it takes a couple of weeks from receiving the bug reports to confirming the fixes. And there are hundreds of commits between the fix-confirmed commit and the vulnerable commit. So, for a specific vulnerability, it's important to locate the commit, where the corresponding fix is applied. Based on the arbitrary reproduce vulnerability with a specific commit hash, we performed the binary search algorithm to locate the fixes. With this method, we can locate the vulnerability patch with log(n) times of reproduction.

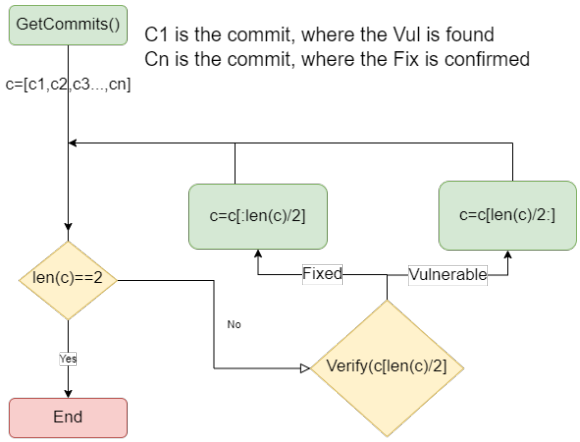

Figure 4: Binary Search to Locate the Fix

As figure 4 shows, we can get the commits where the vulnerability is reported and where the fix is confirmed from OSS Fuzz's report. With these two commits, we can use version control tools to get the commits between these two moments. After that, we can perform binary search on these commits to locating the commit where the fix is applied. The "verify" part in the figure means building the fuzzer target and use the crash input to test if it's vulnerable based on the source code of the middle commit.

5 Framework to Grade Auto-Fixers

In recent years, plenty of AI-based modules can better understand natural language and code. Some of them, such as OpenAI Codex[4], can even take people's instructions and code the program. Also, some researchers use the Codex module to fix the vulerability[5].
Since we can use the method introduced in the previous section to gather vulnerable code and send the code Our database covers kinds of vulnerability and it could be a great source to test auto-fixer such as the Codex module. As figure 5 shows, based on the built database, we can get the vulnerable code and other necessary information about the vulnerability. We assume the auto-fixer would take these data as

Figure 5: Auto-fixer Grading Framework

parameters and fix the vulnerability based on this information.
We can take the auto-fixer as a black box. An excellent auto-fixer would take very vague input, such as a binary file and the source code, and give a bug-free output. This level of auto-fixer has a kind of intelligence and it behaves better than humans. Although this assumption is out of scope, we can see that the more specific information the fixer needs the weaker it is. That also means if we have lower requirements for the auto-fixer, we can give more specific and precise instructions.
In our tentative testing, with human help, we generate as detailed instructions as possible, the Codex module can fix some simple vulnerabilities, including Off-By-One, Use-After-Free, Uninitialized-Variable, and Simple Buffer Overflow.

6 Limitations and Future Work

OSS Reproducer Success Rate About 85% old cases are not reproducible. For projects that committed improper Dockerfile and build scripts, the reproduction is not stable. The key to improving reproducing success rate is asking open-source project developers to write robust Dockerfile and build scripts.

Hard to Evaluate Auto-Fixers For example, the Codex Module, can be used to fix vulnerabilities but it's not supposed to work in this way. Therefore, we should find a way to generate the instruction, which we pass to Codex. And the scale of code that should we send is also hard to decide. Although, large-scale testing can show us several useful.
These two parameters are related to the vulnerability and the complexity of the code.

Evaluation The evaluation part of the OSS Fuzz reproducer and auto-fixer grading framework is not finished because of time issues. I'll attach the evaluation part as well as statistics to this report.

Contribute Code to OSS Fuzz We implemented several meaningful features in this report and people can build the database by running our framework. Also, we plan to wrap the code and give a pull request to merge these features to OSS Fuzz.

Improve Fixes Locator The current implementation can locate 571 cases out of 1279 cases. Although, we didn't fully test all 1279 cases. But we can't perform the binary search algorithm on about 1/3 of cases. Easy improvement might be achieved.

7 Summary

We built a scalable vulnerability dataset based on fresh real-world vulnerabilities and archived the successful cases as a database. With the ability to reproduce vulnerable fuzz targets with a different version of dependencies, we are able to find the commit which fixes the vulnerability. Therefore, the vulnerable source code and its fix are available. With plenty of vulnerable code segments and the corresponding POC, we built a framework to test the auto-fixers.

8 Acknowledgments

I thank Dr. Brendan Dolan-Gavitt for his guidance. He can always give impressive and helpful ideas when I am stuck. Also, I enjoy this research experience with him and would complete this research after the research course. I thank Dr. Hammond Pearce for his help in explaining Codex Model and generating better instructions to guide Codex to fix the vulnerability.

References

[1] "Oss fuzz structure," [EB/OL], https://github.com/google/oss-fuzz/blob/master/docs/images/process.png.

[2] "Oss fuzz report," [EB/OL], https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=54158&sort=-id.

[3] "Oss fuzz reproducer," [EB/OL], https://github.com/google/oss-fuzz/blob/master/infra/build_specified_commit.py.

[4] J. A. Prenner and R. Robbes, "Automatic program repair with openai's codex: Evaluating quixbugs," *arXiv preprint arXiv:2111.03922*, 2021.

[5] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Can openai codex and other large language models help us fix security bugs?" *arXiv preprint arXiv:2112.02125*, 2021.