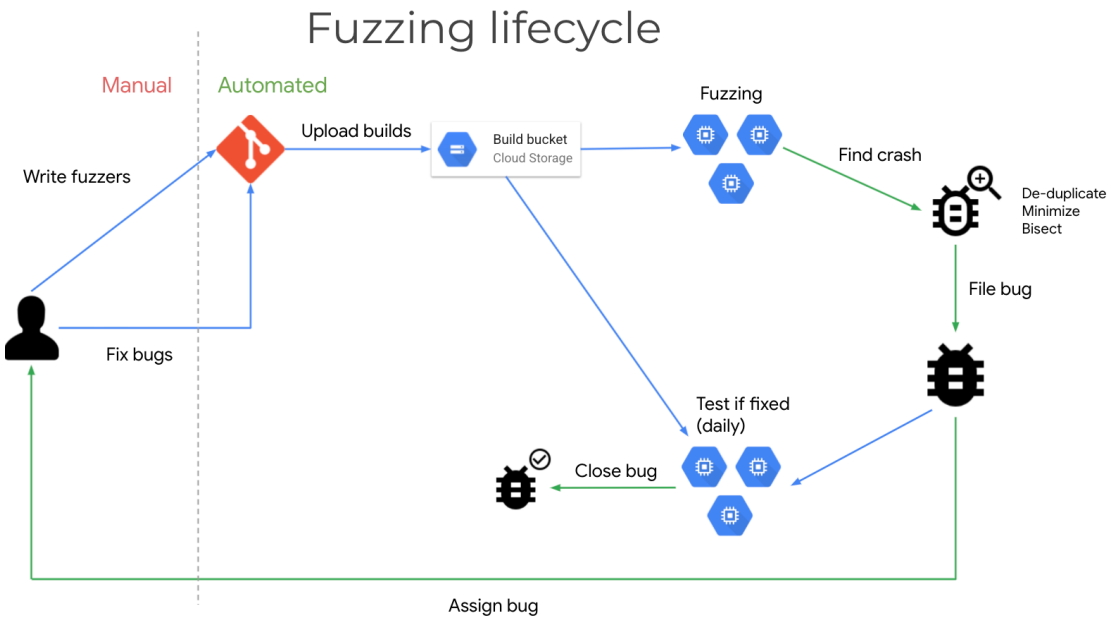
Software Source Code Testing

Submitter: *Eric Brewer, on behalf of Google, LLC*
Topic: *(4) Initial minimum requirements for testing software source code*
Speakers: *Oliver Chang, Staff Software Engineer, ochang@google.com*
Jonathan Metzman, Senior Software Engineer, metzman@google.com

Continuous fuzzing



Fuzzing is the process of feeding automatically generated data to a software program to find unexpected bugs, and it has evolved from a tool used in one-off efforts by security researchers into an integral part of the software development process. Since software is developed continuously and changes over time, fuzzing must also be continuous and work seamlessly alongside traditional software testing practices such as unit or integration testing.

By incorporating continuous fuzzing in development workflows, many security issues can be caught early in the software development lifecycle before any production releases are made. Pre-release fuzzing is especially important because fuzzing is a method also employed by malicious parties; early fuzzing by software developers helps identify vulnerabilities before they are found by others.

Google does fuzzing at scale to great success for almost all software projects. For instance, continuous fuzzing accounts for almost half of all security vulnerabilities reported in the Chromium[1] browser. Beyond Google, services such as OSS-Fuzz[2] also provide continuous fuzzing to the maintainers of over 500 critical open source projects. To date, OSS-Fuzz has resulted in over 5,000 fixed vulnerabilities across projects such as OpenSSL and systemd.

---

[1] Chromium Browser - https://www.chromium.org
[2] OSS-Fuzz - continuous fuzzing for open source software - https://github.com/google/oss-fuzz

The continuous fuzzing workflow is demonstrated in the diagram above. A software developer writes a small fuzzing harness, akin to a traditional unit test. Then, automation will manage the rest of the process end-to-end. This involves continuously building the latest source, to finding crashes via fuzzing, to deduplication and issue filing. Then, after testing shows that the bug has been fixed, the issue is closed automatically, and the cycle repeats itself as the software is further developed. The result is that fuzzing becomes a natural part of all developer workflows.

Throughout this process, artifacts are generated that allow evaluation of the overall success. For example, coverage reports provide a quantitative measure of how much of the software is tested. Fuzzing also continuously builds a corpus of inputs and regression tests that provide another indication of the vulnerabilities that were found and resolved. Turnaround time for bug fixes is another useful measure.

Securing dependencies

Simply testing a software's primary codebase is not enough, as most software relies on a large number of third-party libraries. In addition to fuzzing the codebase, we advocate automatically checking these dependencies for adherence to security best practices and scanning them for known vulnerabilities. The OpenSSF[3] foundation created the Scorecard[4] tool to provide security health metrics for any open source library. This tool can be used to generate an automated report of the security posture of a project and all dependencies that it uses.

For vulnerability scanning and dependency updates, tools such as depandabot and renovatebot provide varying degrees of automation. However, these tools must be accompanied by good automated test coverage to ensure that automated updates can be done reliably without breaking the primary use case.

More development is needed in this space. One key problem today is that matching vulnerabilities to a software's dependencies is difficult to do in an automated way. Many existing standards do not track package names and precise version ranges in a way that is consistent with what's actually used, leading to missed vulnerabilities. There are also many vulnerability databases with different format standards, requiring different tooling and parsers for each. Google has started the OSV[5] effort in collaboration with open source communities to improve automated vulnerability matching and sharing, including a schema format to enable interchange[6].

Summary

To conclude, continuous fuzzing is a very effective way of preventing vulnerabilities from being introduced, and should be integrated into the development process of all software projects. In addition to this, all dependencies of a software project should be regularly checked by automated tools to make sure they follow good security practices and are free of known vulnerabilities.

---

[3] OpenSSF - open source security foundation - https://openssf.org/
[4] Scorecard - security health metrics for open source - https://github.com/ossf/scorecard
[5] OSV - vulnerability database and triage service for open source - https://github.com/google/osv
[6] Proposed vulnerability interchange format for triage automation - https://tinyurl.com/vuln-json