Vorlage für das Konzept

Sebastian Scherer
Daniel Ebert

1 Summary

We want to design and implementation a coverage guided fuzzer for testing client-side javascript running in headless chromium. Our fuzzer is written in python. pyppeteer [6], a high-level API to interact with applications running inside chromium, is used to forward generated inputs from the fuzzer to the client-side javascript and to forward exceptions and coverage information from chromium/client-side javascript to the fuzzer. Our fuzzer is a mutation based greybox fuzzer. Radamsa and mutation strategies from AFL are used to mutate inputs. We focus on finding bugs in the form of unhandeled exceptions, hangs, and excessive memory usage in the client-side javascript.

2 Motivation

In classic web applications, the server calculates and delivers an HTML document to the client. Since a few years there is a trend away from server-heavy code execution to client-heavy code execution. More code means more complexity, which leads to more bugs. The bugs on the client-side must be uncovered quickly and efficiently in order to enable a trouble-free process for the end user. Fuzzers can be used to find these bugs.

Coverage guided fuzzers for testing javascript like Jsfuzz [5] [4] exist already. However these fuzzers focus on testing nodejs packages [5] [4], where javascript code is executed outside the browser. They thus differ from our fuzzer in many areas, especially in the communication between the fuzzer and the client-side javascript and in how coverage is calculated. There are monkey testing tools like gremlins.js [3] and Webmonkey [8] that test web application running inside browsers. These however focus on uncovering bugs via random actions instead of our coverage guided semi-random data approach [3] [8] [2].

3 Fuzzer Workflow

First the user chooses a target. The URL of the target website is passed to the fuzzer. pyppeteer repeatedly opens this website in headless chromium and navigates to the target location in an endless loop. The target location is the UI state where the fuzzer can type in the generated input. Especially in single-page applications, after opening the website

the fuzzer might have to for example press buttons or log in to navigate to the target location. The fuzzer generates an input in every loop iteration. Inputs are generated by mutating randomly selected previous interesting inputs. Previous interesting inputs are inputs generated by the fuzzer that resulted in new code being executed in the client-side javascript. Our mutator is based on radamsa [7] and extended by ideas from AFL. The latter were first mentioned in the AFL whitepaper [10] and described later in more detail in a blog post [9] from AFL's developer Michal Zalewski. Generated inputs are written into one or multiple input elements, for example text fields, via pyppeteer. An example target website can have 2 text fields, one for an email and one for a password. After writing the input, the user can optionally specify actions such as pressing a button or pressing enter. If the client-side javascript throws an unhandeled exception, uses excessive memory, or hangs, the loop iteration is abruptly stopped, the input and information about e.g. the exception is logged, and the fuzzer continues with the next loop iteration. A hang occurs when executing one input takes longer than a specified timeout. Logged inputs are deduplicated using stack hashing [1]. Coverage information of every loop iteration is passed to the fuzzer from chromium via pyppeteer. The fuzzer uses this information to evaluate whether the input is interesting or not.

4 Notizen, kann ignoriert werden TODO

find paper where they talk about how to eval cov talk about bug dedupl

can additionally create a flow diagram optionally scheduling algos from one of the afl 2.0 fuzzers

cleanup includes potentially removing data or cookies from the browser's storage

heap usage via metrics? also stack usage?; unhandeled exceptions via emitters (there is a python package that pyppeteer uses)

Literatur

[1] Valentin Jean Marie Manès et al. „The art, science, and engineering of fuzzing: A survey“. In: *IEEE Transactions on Software Engineering* (2019), S. 13–14.

[2] *Difference between fuzz testing and monkey test*. https://stackoverflow.com/questions/10241957/difference-between-fuzz-testing-and-monkey-test. Accessed: 24.10.2020.

[3] *gremlins.js*. https://github.com/marmelab/gremlins.js/. Accessed: 24.10.2020.

[4] *js-fuzz*. https://github.com/connor4312/js-fuzz. Accessed: 24.10.2020.

[5] *Jsfuzz: coverage-guided fuzz testing for Javascript*. https://github.com/fuzzitdev/jsfuzz. Accessed: 24.10.2020.

[6] *pyppeteer*. https://github.com/pyppeteer/pyppeteer. Accessed: 24.10.2020.

[7] *radamsa*. https://gitlab.com/akihe/radamsa. Accessed: 24.10.2020.

[8] *Webmonkey*. https://github.com/Wildhoney/Webmonkey. Accessed: 24.10.2020.

[9] Michal Zalewski. *Binary fuzzing strategies: what works, what doesn't*. https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html. Accessed: 24.10.2020.

[10] Michal Zalewski. *Technical "whitepaper" for afl-fuzz*. https://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 24.10.2020.