Software Testing, Quality Assurance and Maintenance Winter 2017

Lecture 11 — January 27, 2017

Patrick Lam *version 1*

Fuzzing

Consider the following JavaScript code[1].

```
function test() {
    var f = function g() {
        if (this != 10) f();
    };
    var a = f();
}
test();
```

Turns out that it can crash WebKit (`https://bugs.webkit.org/show_bug.cgi?id=116853`). Plus, it was automatically generated by the Fuzzinator tool, based on a grammar for JavaScript.

Fuzzing is the modern-day implementation of the input space based grammar testing that we talked about in last time. While the fundamental concepts were in the earlier lecture, we will see how those concepts actually work in practice. Fuzzing effectively finds software bugs, especially security-based bugs (caused, for instance, by a lack of sufficient input validation.)

Origin Story. It starts with line noise. In 1988, Prof. Barton Miller was using a 1200-baud dialup modem to communicate with a UNIX system on a dark and stormy night. He found that the random characters inserted by the noisy line would cause his UNIX utilities to crash. He then challenged graduate students in his Advanced Operating Systems class to write a fuzzer—a program which would generate (unstructured ASCII) random inputs for other programs. The result: the students observed that 25%-33% of UNIX utilties crashed on random inputs[2].

(That was not the earliest known example of fuzz testing. Apple implemented "The Monkey" in 1983[3] to generate random events for MacPaint and MacWrite. It found lots of bugs. The limiting factor was that eventually the monkey would hit the Quit command. The solution was to introduce a system flag, "MonkeyLives", and have MacPaint and MacWrite ignore the quit command if MonkeyLives was true.)

[edit: Lecture 12 actually started here]

How Fuzzing Works. Two kinds of fuzzing: *mutation-based* and *generation-based*. Mutation-based testing starts with existing test cases and randomly modifies them to explore new behaviours.

[1] `http://webkit.sed.hu/blog/20130710/fuzzinator-mutation-and-generation-based-browser-fuzzer`
[2] `http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html`
[3] `http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt`

Generation-based testing starts with a grammar and generates inputs that match the grammar.

One detail that I didn't mention last time was the bug detection part. Back then, I just talked about generating interesting inputs. In fuzzing, you feed these inputs to the program and find crashes, or assertion failures, or you run the program under a dynamic analysis tool such as Valgrind and observe runtime errors.

The Simplest Thing That Could Possibly Work. Consider generation-based testing for HTML5. The simplest grammar—actually a regular expression—that could possibly work[4] is `.*`, where `.` is "any character" and `*` means "0 or more". Indeed, that grammar found the following WebKit assertion failure: `https://bugs.webkit.org/show_bug.cgi?id=132179`.

The process is as described previously. Take the regular expression and generate random strings from it. Feed them to the browser and see what happens. Find an assertion failure/crash.

More sophisticated fuzzing. Let's say that we're trying to generate C programs. One could propose the following hierarchy of inputs[5]:

1. sequence of ASCII characters;
2. sequence of words, separators, and white space (gets past the lexer);
3. syntactically correct C program (gets past the parser);
4. type-correct C program (gets past the type checker);
5. statically conforming C program (starts to exercise optimizations);
6. dynamically conforming C program;
7. model conforming C program.

Each of these levels contains a subset of the inputs from previous levels. However, as the level increases, we are more likely to find interesting bugs that reveal functionality specific to the system (rather than simply input validation issues).

While the example is specific to C, the concept applies to all generational fuzzing tools. Of course, the system under test shouldn't ever crash on random ASCII characters. But it's hard to find the really interesting cases without incorporating knowledge about correct syntax for inputs (or, as in the Apple case, excluding the "quit" command). Increasing the level should also increase code coverage.

John Regehr discusses this issue at greater length[6] and concludes that generational fuzzing tools should operate at all levels.

[4] `http://trevorjim.com/a-grammar-for-html5/`
[5] `http://www.cs.dartmouth.edu/~mckeeman/references/DifferentialTestingForSoftware.pdf`
[6] `blog.regehr.org/archives/1039`

Mutation-based fuzzing. In mutation-based fuzzing, you develop a tool that randomly modifies existing inputs. You could do this totally randomly by flipping bytes in the input, or you could parse the input and then change some of the nonterminals. If you flip bytes, you also need to update any applicable checksums if you want to see anything interesting (similar to level 3 above).

Here's a description of a mutation-based fuzzing workflow by the author of Fuzzinator.

> More than a year ago, when I started fuzzing, I was mostly focusing on mutation-based fuzzer technologies since they were easy to build and pretty effective. Having a nice error-prone test suite (e.g. LayoutTests) was the warrant for fresh new bugs. At least for a while. As expected, the test generator based on the knowledge extracted from a finite set of test cases reached the edge of its possibilities after some time and didn't generate essentially new test cases anymore. At this point, a fuzzer girl can reload her gun with new input test sets and will probably find new bugs. This works a few times but she will soon find herself in a loop testing the common code paths and running into the same bugs again and again.[7]

Fuzzing Summary. Fuzzing is a useful technique for finding interesting test cases. It works best at interfaces between components. Advantages: it runs automatically and really works. Disadvantages: without significant work, it won't find sophisticated domain-specific issues.

Related: Chaos Monkey

Instead of thinking about bogus inputs, consider instead what happens in a distributed system when some instances (components) randomly fail (because of bogus inputs, or for other reasons). Ideally, the system would smoothly continue, perhaps with some graceful degradation until the instance can come back online. Since failures are inevitable, it's best that they occur when engineers are around to diagnose them and prevent unintended consequences of failures.

Netflix has implemented this in the form of the Chaos Monkey[8] and its relatives. The Chaos Monkey operates at instance level, while Chaos Gorilla disables an Availability Zone, and Chaos Kong knocks out an entire Amazon region. These tools, and others, form the Netflix Simian Army[9].

Jeff Atwood (co-founder of StackOverflow) writes about experiences with a Chaos Monkey-like system[10]. Why inflict such a system on yourself? "Sometimes you don't get a choice; the Chaos Monkey chooses you." In his words, software engineering benefits of the Chaos Monkey included:

- "Where we had one server performing an essential function, we switched to two."
- "If we didn't have a sensible fallback for something, we created one."
- "We removed dependencies all over the place, paring down to the absolute minimum we required to run."
- "We implemented workarounds to stay running at all times, even when services we previously considered essential were suddenly no longer available."

[7] `http://webkit.sed.hu/blog/20141023/fuzzinator-reloaded`
[8] `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`
[9] `http://techblog.netflix.com/2011/07/netflix-simian-army.html`
[10] `http://blog.codinghorror.com/working-with-the-chaos-monkey/`