Interview with Natalie Silvanovich

RIK FARROW

Natalie Silvanovich is a security researcher on Google Project Zero. Her current focus is browser security, including script engines, WebAssembly, and WebRTC. Previously, she worked in mobile security on the Android Security Team at Google and as a team lead of the Security Research Group at BlackBerry, where her work included finding security issues in mobile software and improving the security of mobile platforms. Outside of work, Natalie enjoys applying her hacking and reverse-engineering skills to unusual targets and has spoken at several conferences on the subject of Tamagotchi hacking. natalie@natashenka.ca

Rik is the editor of *;login:*. rik@usenix.org

I met Natalie Silvanovich at the luncheon during USENIX Security '19 in Santa Clara. We had a fun discussion, and I resolved to spend some time following up later.

Rik Farrow: I am familiar with a really "old" way of finding bugs: fuzzing. I know this was very common in the late '90s, and I assume you were using fuzzing sometimes when you worked at BlackBerry. What's different about how you search for bugs today?

Natalie Silvanovich: It's been nearly 15 years since I started doing vulnerability research, and in some ways the fundamental techniques for finding security bugs haven't changed much. Fuzzing and code review (or binary analysis for software that doesn't have source code available) are still the techniques I use to find the majority of bugs I report. What has changed is the maturity of each methodology.

There have been a lot of tools and techniques developed over the past few years that have greatly improved the efficiency and effectiveness of fuzzing. I think one of the most important innovations is fuzzers like AFL (http://lcamtuf.coredump.cx/afl/) that use code coverage measurements to guide fuzzing, so that the fuzzer can focus on testing new and unexplored areas of software. Also important are tools that allow for fuzzing to be performed at scale, for developers to easily integrate fuzzing into the development process, and for errors to be more consistently detected when fuzzers hit them.

The flip side of this is that, in general, it is more difficult to find bugs with fuzzing these days. I think this is due to more security awareness among developers, as well as more software teams fuzzing their code as a part of the development process. Fifteen years ago, it was common to find security bugs using simple mutation fuzzing on a single host in a few hours. Now it usually takes more advanced techniques on multiple cores.

Code review techniques have been fairly consistent throughout the years, although now, of course, we know a lot more about bug classes and how attackers can exploit them. It is also generally more challenging to find security bugs with code review these days, probably because software is both better tested and more complex.

RF: As part of Project Zero, do you ever work as groups/teams on a project?

NS: Yes, we do. In fact, I worked on a large research project on the iPhone [1–5] with Samuel Groß last year. We also do team hackathons a few times a year where we work on the same target together. While we do a lot of independent research, there's a lot to be gained by sharing ideas!

RF: Do you and others in Project Zero get direction on what software to search, or can you pick and choose?

NS: Project Zero's mission is to "make zero-day hard," and we pick our targets based on this mission. Usually, this means software that has a large user base, a history of being targeted by certain attackers, and/or a vulnerable user base. Team members are free to pick their own targets within the mission, although we also often discuss targets and make goals as a team.

RF: You wrote about what someone should do to get hired at Project Zero (https://googleprojectzero.blogspot.com/p/working-at-project-zero.html), and I wondered if you have thought of anything you'd like to add since you posted that?

NS: Not really, but I would like to mention that vulnerability research is just one of the many careers available in information security, and that post is very specific to our team.

RF: Are there other women working on the Project Zero team? In my experience, the number of women working in security is even lower than in other areas in IT—much lower.

NS: I was the only woman on Project Zero for about four years, but we've recently been joined by the amazing Maddie Stone. There are fewer women in information than a lot of other IT fields, but it's improved somewhat over the last few years.

RF: With your goal of making zero-day hard, I wonder what things you consider can make security better. I find myself surprised that things have gotten better, as most programmers are average in skill, and the languages they most often write in, C and C++, are the same as they were when they were first created when it comes to security. For example, a programmer can still use `gets()` on Linux, and buffer overflows are still possible, although compiler support for protecting the stack has pushed their exploitation to the heap.

NS: This is a huge question, because there are so many ways to improve software security. And I also want to qualify "things getting better"—while I suspect there are fewer bugs per line of code today than there were in the past, there is also more software being used by more users for more applications than ever before. So overall, software security is a more important problem than it has ever been.

Taking the example of a call to `gets()` that causes an overflow, there's a lot of things that can happen during the development process that can stop it from getting into release code. For example:

- The developer understands that `gets()` can lead to vulnerabilities, and doesn't use it.
- The developer's compiler or development environment warns them about `gets()`, and they remove it.
- The repository they submit the code to has pre-submit or compiler checks that reject `gets()`, and the developer can't submit their code until they fix it.
- Submitting code requires the commit to be reviewed by another developer, and that developer finds and fixes the bug.
- The code in the repository is automatically fuzzed, and the bug gets found before release.
- The code is security reviewed before it is released, and the bug gets found before release.
- The crash occurs during beta testing, and the developer fixes it based on the crash log.
- The release binary contains mitigations that make it more time-consuming to exploit memory corruption bugs.

Good "development discipline" can greatly reduce the number of security (and other) bugs in software, and there are a lot of tools and technology available to help with this. Of course, this requires that the organization produce the software to prioritize and invest in security, which is unfortunately not always the case.

RF: While I am still a fan of LangSec (langsec.org), I now realize that it is just a part of the overall picture of secure programming practices. What do you think of LangSec, and where do you see that LangSec falls short of what programmers need to be doing?

NS: LangSec aims to improve software security by creating formally verifiable languages and parsers that are immune to many common security problems. They view the root cause of security issues to be that most protocols and other input formats are poorly defined and often have many undefined states, and the programming languages that process them also support a huge amount of undefined behavior. They think all software should abstract out all input processing code, and design and implement it in a way that is verifiable and has no undefined states or behavior.

One observation behind LangSec's philosophy is that the language software is written in has a huge influence on the number of vulnerabilities it contains. There is a lot of evidence for this. The most important distinction in my mind is managed (does not allow dynamic memory allocation) versus unmanaged (allows dynamic memory allocation) languages. Since the majority of vulnerabilities exploited by attackers are memory corruption vulnerabilities that occur due to the misuse of dynamically allocated memory, even just moving to dynamic languages has a lot of potential to reduce the number of vulnerabilities in software.

LangSec's goal is lot broader than increasing the use of managed languages, though. Dynamically allocated memory is just one of the causes of the undefined and unverifiable software behavior they want to prevent. Unfortunately, while there would be a lot of benefits to fully verifiable input processing, the reality is that technology is not quite there yet. Even just with managed languages, there are a lot of reasons that developers don't use them, including performance, capabilities, and compatibility with legacy code, and formally verifiable languages have even more limitations. So while LangSec's ideas are very promising for the future, I feel that a lot more work needs to be done before their work is practical for most applications.

Another concern is that LangSec's approach doesn't prevent logic bugs. For example, imagine a shopping website that notifies the warehouse to ship an item before it collects payment. This design has a security problem where if a user gets to the point where the shopping service notifies the warehouse to ship, and then the user stops interacting with the site, the user will get the item for free. Formal verification won't prevent this type of problem, it will only check that the implementation conforms exactly to the design. It is also likely that any formally verifiable language or parser has at least some bugs in it (because all software has bugs), which could lead to security bugs in software that uses that language. It's also possible attackers think of new types of vulnerabilities that no one has thought of yet. So while LangSec's approach would likely greatly reduce the number of vulnerabilities in software, it won't eliminate all of them.

That said, there are two important takeaways from LangSec's approach that developers can use right now. One is that the language they choose to write software in impacts its security a lot. The other is that design is really important. The better defined a feature is, and the more thought that is given to making it easy to implement securely, the more likely it is to be secure.

RF: Other than good "development discipline," what else can programmers prevent to make their software more secure?

NS: One important strategy for improving software security is Attack Surface Reduction. Put simply, every piece of software has a portion of code that can be manipulated by attackers, and making this as small as possible can have huge returns with regards to preventing vulnerabilities. It's not unusual for Project Zero to find bugs in software features that have low or no usage, meaning they present security risk to users with little benefit. It's important for developers to be aware that all code creates a security risk and other bugs, and to make sure that tradeoff makes sense.

References

[1] The Fully Remote Attack Surface of the iPhone: https://googleprojectzero.blogspot.com/2019/08/the-fully-remote-attack-surface-of.html.

[2] The Many Possibilities of CVE-2019-8646: https://googleprojectzero.blogspot.com/2019/08/the-many-possibilities-of-cve-2019-8646.html.

[3] Remote iPhone Exploitation, Part 1: https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-1.html.

[4] Remote iPhone Exploitation, Part 2: https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-2.html.

[5] Remote iPhone Exploitation, Part 3: https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-3.html.