EDITORIAL

Musings

RIK FARROW

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

While musing, I like to wonder what it would be like to live in a world without buggy software. That is, a world very unlike the one we live in. As I write this, Boeing's 737 MAX plane has been grounded, apparently because buggy software and not documenting its possible dangerous effects have killed over 300 people in two separate crashes. Businesses and home users regularly have their data encrypted by criminals demanding ransom. And whole countries are in turmoil via careful manipulation of opinion via social media.

I attend conferences looking for people with interesting and potentially useful ideas. I first met Kostya Serebryany at Enigma 2016, where I tried to get him to write about the work he has been doing in security. He deferred then. Kostya then contacted me in the Fall of 2018 excited about something I find exciting as well: adding security features to hardware. We've published articles from several authors about hardware features to improve security, as well as problems with hardware solutions, such as the ability to extract data from Intel's secure enclave, Meltdown [1].

Kostya most recently has worked on fuzzing, techniques for probing programs for potentially exploitable bugs. In 2015, Peter Gutmann wrote about various fuzzing techniques, something that Kostya has long worked on, and that's related to what he wrote about for this issue [2].

Weaknesses in C/C++

I've long joked that C was a macro-assembly language: a convenience layer for those who needed to write code near to the speed of assembly [3], but with the convenience of variable labels, `for` loops, subroutine call handling, and structures. When I first encountered C, I immediately fell in love with structures, as the concept made some of the things I needed to do so much clearer than calculating offsets in assembler would have been. And, to be honest, I was really bad at calculating offsets. C beat the hell out of writing in Intel assembly (or VAX or Motorola assembler too).

But C and C++ lack certain safety features found in modern languages like Java, Go, Swift, and certainly Rust. In C and C++, you could specify array indices far beyond the end of the array you'd locally allocated, leading to buffer overflows on the stack. You could do this as well in the heap, and you could also do this with pointers into memory. I consider C and C++ to be languages for expert programmers, because they made it so easy to do the wrong thing. I always assumed that the authors of these languages were highly intelligent and expert programmers themselves, and that they had written these languages for their own convenience. In the case of C, that was certainly true, although the authors would be sharing C with other Bell Labs employees and, eventually, professors at various universities.

Bjarne Stroustrup, also at AT&T Bell Labs, came along a bit later, added classes to C, but kept all its wonderful and dangerous flaws. That is, you could create classes and instantiate objects, but you could also overrun arrays, leak memory, and abuse pointers.

Smashing the Stack

The Internet Worm really made people aware of the danger of buffer overruns. The `finger` daemon used the C function `gets()`, which collects a string into an array previously allocated but doesn't check to see whether the length of the array is sufficient. This function still exists in `libc`, and the man page includes the warning, "Never use this function." Makes you sort of wonder why it's still there.

I learned much more about smashing stacks from Elias Levy's famous article about buffer overflows [4]. I recreated the `finger` daemon for class exercises and gave students short C programs they could use to attack the `finger` daemon, whose real purpose was to run the `who` command and return the results over the network. When correctly exploited, the attack would instead run `/bin/sh`.

And this was only part of the problem with C and C++. There were also ways to exploit file structures that contain pointers to functions, or to use a little known option of `format()` to carefully overwrite portions of the stack, allowing exploits that used Return Oriented Programming (ROP). And this is just a partial list.

There are other issues with C/C++ that have to do with pointers. Using `malloc()` returns a pointer to a block of memory, and `free()` releases that block. But it's quite common for programmers to either forget to free memory (a memory leak) or to use a pointer to memory after it had been freed (use-after-free).

During the first time I met Kostya, he showed me dozens of places in the Linux kernel where memory was used after it was freed and was still unpatched upstream. I could tell he was agitated about this.

Today C and C++ are the second and third most popular programming languages (as of April 10, 2019) in the Tiobe Index [5]. Looking at language popularity in another way, I asked Chris Wysopal of Veracode about how many programs in various languages that they analyze each year, and Chris provided me with the diagram in Figure 1. Veracode's numbers, based on the thousands for binary programs analyzed, present a different picture, where C/C++ is less popular.

I found myself wishing that C would just go away, but Kostya assured me that that's not going to be happening, as IoT devices will use slower CPUs and have less memory, and they are going to need compact and fast languages. Damn.

The Lineup

Jasmine Peled, Bendert Zevenbergen, and Nick Feamster have written a column about ethics, regarding something I had never heard of, called mcTLS. You might think that something with *TLS* in its name has to do with encrypting Internet traffic, and you'd be right. However, mcTLS has to do with creating a method

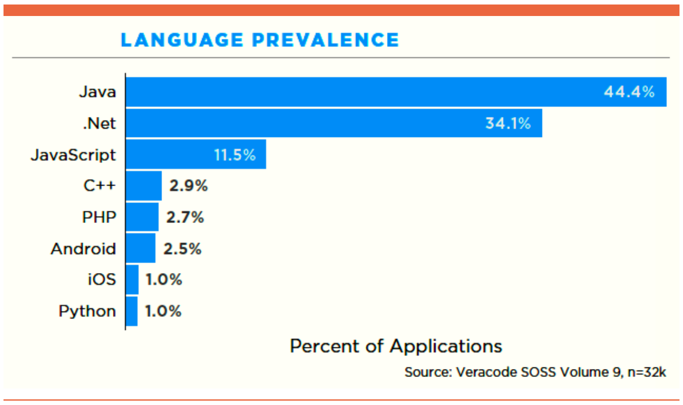

Figure 1: Popularity of programming languages based on programs analyzed for vulnerabilities by Veracode

so that TLS can be decrypted by middle boxes. If you think this is a bad idea, Peled and her co-authors agree with you, and explain why even the initial researchers should have considered this. Note that the IETF isn't happy about mcTLS either, mainly because including *TLS* in the name violates copyright as well as having the ability to confuse people about their Internet traffic actually being secure.

Kostya Serebryany has written about a security extension in hardware, something I consider a wonderful idea (in case you skipped the earlier part of this column). Sun, now part of Oracle, first came up with the notion of including tags to help prevent a variety of bugs and the successful exploitation of those bugs, and now ARM plans on doing this as well.

I interviewed Mark Loveless, aka Simple Nomad. I've known Mark for many years, and we got together during Enigma '19 to chat and begin this interview. Mark is definitely someone you should call a hacker, unlike Beto O'Rourke, whose membership in the Cult of the Dead Cow predates most of the cDc's hacking activities. Mark has interesting stories to tell.

Anuj Kalia, Michael Kaminsky, and David Andersen have written about eRPC. You might recognize the authors' names from an earlier article about RDMA. This article, like the first one, is based on a paper, this time at NSDI '19. While their paper takes a deeper dive, Kalia et al. explain how this open source RPC library can be faster than those that rely on niche networking technologies.

Daniel Bittman, Peter Alvaro, Darrell Long, and Ethan Miller write about how to avoid bit-flipping in programming data structures. Based on a FAST '19 paper, Bittman et al. explain why bit-flipping may be considered harmful for persistent memories, like Micron's XPoint. But what I particularly like about their work is that it offers a different way of thinking about, and using, traditional data structures like linked-lists and B-trees that is often faster—and involves smaller structures and fewer bit flips.

Vladimir Legeza and Anton Golubtsov tell us how to make logging much more useful. Legeza, now working at Google, and Golubtsov (Amazon) suggest what should be commonsense methods for having standards for your logging messages. Legeza first suggested this idea as an opinion article, but I consider it much more along the line of best practices. I wish I had read such an article 35 years ago!

Laura Nolan considers complexity, taking a different perspective from Dave Mangot's "Boring Tech" article [6] in the Spring 2019 issue. Laura first describes what is meant by software complexity, then how systems complexity differs from the software version. Laura does a great job, and she has volunteered to write columns about SRE issues.

Peter Norton has written about how you can use a tool based on Python to create portable configuration files. The external format is YAML, and the code performs static type checking, helping to prevent errors in configuration.

Mac McEniry decided to cover the use of password managers. Mac has previously written about Hashicorp's Vault (Winter 2017) [7], but this time around he looks at three different Go libraries for secure storage of passwords for use by applications: Keychain (Mac), Windows Credential Manager, and a library called `keyring` that will work on Linux and the other OSes as well.

Dave Josephsen considers just how weird and wonderful it is to be living in the middle of nowhere in Montana. Then Dave gets down to business and begins explaining why he likes Prometheus for monitoring so much and how it's used.

Dan Geer ponders about just how common exploited software bugs might be. Working from various data sources, Dan tells us that the problems with software bugs are much worse than you likely suspect, and even worse than I imagined.

Robert Ferrell suggests that we tone down our expectations for technology. After all, flying cars are still experimental, and even Amazon has decided that having a special button just for ordering laundry detergent might not be the best use of technology.

Mark Lamourine has written three book reviews, covering *Refactoring* (second edition), *Concurrency in Go,* and *Cloud Native Go.* I reviewed David Clark's *Designing an Internet*, and also wrote two short reviews of books for summertime reading: Marcia Bjornerud's *Timefulness* and Max Gladstone's *Empress of Forever.*

In Closing

There are problems with all programming languages. For example, while Rust is much safer by design, you can write Rust code in unsafe mode, disabling its safety features. Java does checks and prohibits array overruns, but the JVM is written in C++, and it has had numerous vulnerabilities over the years.

I also asked Chris Wysopal if he could tell me what proportion of exploitable bugs came from code that processed input, and he answered 75%. If you've been reading *;login:* for the last five years, you will have noticed, and hopefully read, many articles relating to LangSec, for example [8, 9]. LangSec, roughly, is the notion that security could be tremendously improved by paying more attention to input parsing, and Chris's comment about the majority of vulnerabilities coming from input parsing problems supports this.

When I heard about LangSec and learn about efforts to create better support for security in hardware, I imagine that the problem of software insecurity will soon be solved. But I am forgetting several things.

First, most programmers are, by definition, of average skill level. Second, few programmers know much about security, and far fewer have a clue about LangSec. Third, some protocols, like the text (versus binary) version of X.509 certificates, cannot be parsed securely because the design requires a complex parser. And finally, even when ARM or Intel produce security features that will greatly reduce successful exploits, most people won't enable them, either because they don't understand them or because such features cause programs to fail sometimes—an indication of programming flaws they'd prefer to ignore.

References

[1] D. Gruss, D. Hansen, B. Gregg, "Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer," *;login:*, vol. 43, no. 4 (Winter 2018): https://www.usenix.org/publications/login/winter2018/gruss.

[2] P. Gutmann, "Fuzzing Code with AFL," *;login:*, vol. 41, no. 2 (Summer 2016) : https://www.usenix.org/publications/login/summer2016/gutmann.

[3] Wikipedia, "Assembly Language: Macros," last modified on March 25, 2019: https://en.wikipedia.org/wiki/Assembly_language#Macros.

[4] E. Levy, "(Aleph One), Smashing the Stack for Fun and Profit," *Phrack,* vol. 7, no. 49: http://phrack.org/issues/49/14.html.

[5] Tiobe Index, April 2019: https://www.tiobe.com/tiobe-index/.

[6] D. Mangot, "Achieving Reliability with Boring Technology," *;login:*, vol. 44, no. 1 (Spring 2019): https://www.usenix.org/publications/login/spring2019/mangot.

[7] C. McEniry, "Go: HashiCorp's Vault," *;login:*, vol. 42, no. 4 (Winter 2017): https://www.usenix.org/publications/login/winter2017/schock.

[8] S. Bratus, M. Patterson, and A. Shubina, "The Bugs We Have to Kill," *;login:*, vol. 40, no. 4 (August 2015): https://www.usenix.org/publications/login/aug15/bratus.

[9] J. Bangert and N. Zeldovich, "Nail: A Practical Tool for Parsing and Generating Data Formats," *;login:*, vol. 40, no. 1 (February 2015): https://www.usenix.org/publications/login/feb15/bangert.