Musings

RIK FARROW

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

I first encountered Sergey Bratus in a dingy stairwell in a Westin hotel in San Francisco. We were attending the 20th USENIX Security Symposium, and Sergey was a co-author of two WOOT papers. I could tell that an article pitch was coming and listened carefully as Sergey expounded on *weird machines*, unplanned-for VMs that exist in most code.

Sergey's student, James Oakley, had won the Best Student Paper award for showing how `gcc`'s exception handling format (DWARF) was rich enough to provide a complete execution environment. While this notion appeared a bit obscure to me, even as it was alarming that DWARF was exploitable, I still wondered just how big the impact was. I have paid attention to new exploits since I became interested in UNIX security in 1984, and couldn't recall any exploits that relied on this particular format.

Sergey, a short, rounded man with a graying comb-over, patiently explained to me that it wasn't just this example: weird machines could be found everywhere in code. And the more Sergey talked, the more I began to see the connection between the exploits I had studied over many years and what he explained in that dim and echoing stairwell.

Sergey, a Research Associate Professor at Dartmouth, has co-authored many papers and several *;login:* articles on this topic since that day. He approached me again this year (by email), asking me if I wanted to attend the LangSec workshop [1] happening as part of the IEEE Security and Privacy Workshops in May 2015. He also had another article idea, but I wanted something different: a clear description of the problems caused by weird machines, without resorting to insider jargon (like the term *weird machines*). Fortunately for us, Sergey, along with Meredith Patterson and Anna Shubina, did spend a lot of time writing an article for this issue. And I believe they've done a great job.

If I were to attempt to describe this issue as an elevator pitch (you have just 30 seconds), here's what I'd say. There is a programming issue that is the single cause of most exploits, and while it is possible in many cases to fix this problem, it has been ignored. This issue can be fixed by using programming techniques, many over 40 years old, that get ignored by programmers who write exploitable code instead. But there are cases where proper coding cannot help you, because the protocols involved are too complex by design. And some of those impossibly complex protocols include some of the foundations for the security of the Internet, like TLS and HTML5.

While fixing problems with input parsing, the appropriate place in any program, won't solve all security issues, this single type of fix would do more to improve the security of our computers, cars, smartphones, and devices than would any other change. In fact, any software-controlled device that accepts input beyond a simple on-and-off switch will *never* be secure without observance of the principles described in Sergey's article. Those principles are based on both research as well as years of observation into exploitable software, and the conversion to having parsers that can be proven to be correct will have more impact than anything else we could possible do to improve security today.

Our computers are flexible by design—that's what makes them so useful for doing a huge variety of tasks. If we expose our computers through the use of complex parsers or protocols to Turing-complete input languages, we must expect that our software, and our devices, can never be made secure. Attackers will continue to make our devices dance.

The Lineup

I've already provided an introduction to the first article, so let's consider the second. Sun et al. wrote a HotOS workshop paper about their research into unreliable operating systems. Their insight is that many exploits are brittle, and providing some randomness to the responses of the operating system to programs that aren't whitelisted will disrupt their behavior.

Zhuang et al. have built an environment that supports the collection of sensor data from smartphones. Their solution must overcome both privacy concerns and security issues involved in running software on strangers' phones.

I interviewed Marc Maiffret, a self-educated man who founded a successful security company at age 17 after a bit of a rough start. Marc has a unique viewpoint into the world of Microsoft security, having helped to prod Microsoft into a better security posture.

Mark D. Roth explains how Google uses an unreliability budget to provide more reliable services. This is a neat idea, one I first heard about during SREcon in 2014, and am happy that the unreliability budget has finally been clearly explained.

Andy Seely continues his series on managing with an article looking at the seven levers that can be used to help retain talented employees.

Gunawi et al. have shared their ongoing research into the causes of failures in distributed applications, such as HDFS and Cassandra. Some of the problems only appear at large scale, making them difficult to test, while others are more tractable.

David Beazley continues his two part series on concurrency in Python by explaining coroutines. Coroutines rely on application-level programming to provide a form of concurrency, using `yield`, but still have the Global Interpreter Lock to deal with.

David N. Blank-Edelman also has a second part in his own series about concurrency in Perl, using the Coro modules. Coro uses `cede` to yield control to other threads, and this can be done using semaphores, or by using other modules, like AnyEvent.

Dave Josephsen shares his experience in determining Key Performance Indicators (KPI), in particular, by choosing the latencies measured between the components of a service.

Dan Geer and HD Moore have taken a measured look at the number of IPv4 addresses that you can actually probe, and it appears that there are huge enclaves of devices that are hidden, generally by mobile broadband providers. There are also, of course, devices that we wish *were* hidden, provided mostly by cable companies.

Robert G. Ferrell muses about the future of quantum computing. Specifically, just how will we write scripts to manage systems where each test value can be both true and false at the same time.

Mark Lamourine has written two book reviews for this issue. His first covers a book on Swift, Apple's new language for apps. Mark takes a look at a book on programming in Python on the Raspberry Pi for his second review.

I started this column discussing a topic, input parsing, that is actually not as simple as I might have implied. I doubt that many programmers today have even heard of the Chomsky hierarchy of formal languages [2], first described by Noam Chomsky in 1956. And even if programmers are aware of this hierarchy, grasping the difference between context-free and context-sensitive grammars will be far beyond what we should expect of people writing Web applications in PHP or JavaScript.

But I certainly believe that computer scientists and members of industry who are responsible for protocols, such as HTML5, TLS, X.509, XML, and IPv6, should be aware of the implications of designs that require nondeterministic Turing machines, that is, ones that cannot be proven to be correct, to interpret them. When we base our technological future on systems that are insecure by design, we should not be surprised by that very lack of security that surfaces daily.

References

[1] Second Workshop in LangSec (Language Security): http://spw15.langsec.org/; first workshop: http://spw14.langsec.org/.

[2] The Chomsky Hierarchy: https://en.wikipedia.org/wiki/Chomsky_hierarchy.