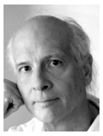
OPINION

Musings

RIK FARROW



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

The latest effort to improve computer and network security has been for the US Congress to pass new laws. Currently proposed laws will increase penalties for attackers, including the perhaps misguided hackers who point out easy-to-find flaws in public servers, to the point of treating them like racketeers. The real culprits, the companies who leave treasure troves of information with real worth easily accessible, will be encouraged to do better. I think there are better solutions.

And for a change, this issue actually includes an article that points the way forward to improving network security. I have whined way too often about how bad things are, giving the appearance that I'm terribly depressed, so I am happy to present something much more useful than more complaints about the state of things.

The Exploit Issue

The first three articles in this issue deal with exploits, the methods used for attacking server and client software. You might not think that reading more about exploits will result in techniques for preventing them, but please give us a moment of your time.

We begin with an article by Ed Schwartz. Ed presented a paper at USENIX Security '11 on Q [1], software that finds short code segments that can be used in one type of exploit. I wanted Ed to write for *;login:* because he did an outstanding job of explaining return-oriented programming (ROP) during his Security presentation. Ed's excellent article starts by explaining how different exploits work, two of the current countermeasures, and why they do not always succeed. He also provides a first-rate discussion of both past and current methods for taking control of the flow of execution, then executing the code of the attacker's choice. As Ed writes, "At a high level, all control flow exploits have two components: a computation and a control hijack. The computation specifies what the attacker wants the exploit to do. For example, the computation might create a shell for the attacker, or create a back-door account. A typical computation is to spawn a shell by including executable machine code called shellcode."

And as Ed has pointed out, the attacker wants to do something that the target software was neither designed nor written to do. Instead, the attacker needs to use existing mechanisms in the software, and its supporting libraries, to do something completely unexpected.

Next, the paper by Sergey Bratus and his co-authors ties in quite beautifully with Ed's article. I first met Sergey Bratus during WOOT '11, where he was explaining how code found in every Linux executable actually includes a complete Turing machine that can be abused [2]. Sergey called this code, and examples like it, *weird machines*, and I find I like this terminology. A weird machine provides an attacker with the ability to execute his own code, completely contrary to the intentions of the software's designers. Yet these weird machines must be present for this to work. And we have hundreds, if not thousands, of existing exploits that prove that these weird machines actually exist.

Sergey also acted as the point person for the third article in this collection. He had to, because its lead author, Len Sassaman, died this summer, before the article was proposed. Len, his wife, Meredith Patterson, and others had been working on a paper that looked at exploits in a different light. What makes exploits work, besides having weird machines to run them on, are the inputs to those weird machines. The authors' proposition was that every program that accepts inputs has its own input language. If that input language reaches beyond a minimal level of complexity, it is impossible to prove that the program that parses the input language will behave as expected. Instead, the program will be one of the weird machines that run exploits for attackers.

The Sassaman proposal has its basis in formal language theory, where a program's input forms the language and the program includes the parser for this language. We are all familiar with this concept, whether we have written programs or simply entered command lines with options. The options make up the input language, and the program must include a parser that interprets that input language. This example may sound too simple, but there have been command-line programs that were exploitable. And network servers have much more complex input languages, with databases supporting SQL perhaps near the pinnacle of complexity. The authors do a fine job of arguing this point.

When Sergey first described this idea to me, in the hallway at Security, I made an immediate connection to an early, and somewhat effective, security prophylactic: application gateways. Application gateways were used, usually as part of firewalls, to parse the input to the services they protected. For example, the application gateway for the SMTP protocol follows the RFC for the protocol exactly (RFC 821, when smapd [3] was written). One result of the slavish adherence to protocol specifications was that crafted inputs from exploits that violated the protocol *in any way* were prevented from reaching the mail server. If the exploit relied on email addresses that were longer than 256 bytes or message lines longer than 1024 bytes, these inputs would be blocked. This is how application gateways could block never-before-seen attacks: they only accepted a very precisely defined input language and rejected all others.

Although application gateways are uncommon today, their close relatives are still in use. Web application firewalls may either enforce a well-defined input language or act more like an Intrusion Prevention System by watching for and blocking known attack signatures. But these were just perspectives that I had when I began talking with Sergey.

Sassaman et al. lay out three points that can be used as a test of their theory's usefulness. They also contrast programming languages, which have input parsers derived from a formal grammar, to most programs, whose input parsers are improvised to support specifications—not nearly as robust a process. And once the input language has grown behind a very simple level of complexity, the parser for the language can become Turing machine complete: itself a complete computer for which the halting problem can never be decided.

If the suggestions in Sassaman et al. were perfectly understood and universally adopted, they would not end exploitation. There would still be bugs in implementation that a simple and provably correct input parser could not protect from exploitation. But the attack surface would be greatly diminished by keeping the input simple, and thus easier for programmers to understand and for designers to specify. And there will always be some input languages (JavaScript and X.509 are examples) which can never be made secure from crafted inputs.

More Security

Let's shift gears a bit, as I describe the next article in the lineup. Adam Langley works for Google as a Chrome developer. I didn't meet him in San Francisco during Security, but I did get to read the summary of the panel on SSL/TLS certificates. Given the recent event where a Dutch certificate authority was compromised [4] and their signing key used to produce bogus and yet totally valid certificates, I thought it would be useful to have someone who could write about this issue.

Adam surprised me by writing about how to properly configure Web servers, when I thought he might be writing about browser insecurity. Yet he is in a very good position to be talking to Web server administrators, as he knows what the browser developers (including Mozilla Firefox) have done or plan to do. Adam is also aware of issues impacting browser users that can be mitigated through the correct use of HTTPS, as well as the proper design of Web pages that will be served over HTTPS.

Adam also discusses the potential for DNSSEC and certificate verification, something that I had really hoped he would do. Paul Vixie's article about DNSSEC in the October 2011 issue had gotten me more interested in the potential for DNSSEC to help with the complex issue of certificate authorities, and Adam presents a clear and direct browser developer's perspective.

Sliding more directly into the intersection of sysadmin and security, Jan Schaumann tells us about a tool he wrote while working at Yahoo! that has since been released as open source. Jan's tool, sigsh, allows the execution of shell scripts that have been signed by authorized users. The signature verifies both the integrity of the script and that the script itself has been authorized for use. Just by maintaining the public key file found on the servers—in this case, many thousands of distributed systems—arbitrary commands can be executed safely for the purposes of system maintenance.

Patrick Debois completes the lineup with an article about DevOps from the sysadmin's perspective. As DevOps is LISA '11's theme, I wanted to understand more about it. And Patrick, as one of the standard bearers for this movement, seemed like just the person to write about DevOps. Patrick does a great job of explaining both the motivation behind DevOps and the goals that can be achieved through its use.

Columns

David Blank-Edelman explains how you can invoke Perl from within your favorite text editor and perform useful, Perl-related tasks. I had often wondered if David just might be a bit obsessive, because of the way he would line everything up *just so* in his code examples. It turns out he wasn't wasting time, but was using Perl modules invoked from within his editor of choice to beautify his code. David also explains other tools you can use. Thanks, David, as I for one am relieved.

Dave Josephsen has found a new suite of system monitoring tools, Graphite, that has got him excited. Graphite is a game changer, writes Dave, and consists of three Python programs: Whisper is a reimplementation of a round-robin data program, Carbon collects data from the network and writes it to Whisper, and Graphite (with the same name as the suite) provides the Web front end. Dave promises to write much more about the elegant solution provided by Graphite in future columns.

Robert Ferrell decided to stick to writing about exploits. Well, sort of. Robert engages in a search for the real meaning of "exploit," and the outcome is as unpredictable as ever.

Elizabeth Zwicky tells us about the books she has been reading, including two that consider what motivates people—in this case, two very distinct groups of people. She also reviews a book about software quality and statistics (another of her favorite topics). Sam Stover, meanwhile, got really excited about the new Kevin Mitnick book. While Mitnick will always be a villain in many people's eyes, Sam loved the way Mitnick and his co-author manage to tell stories with many technical details while taking the reader for a great ride. I don't expect that this book will change anyone's feelings about Mitnick, but it will certainly educate anyone who reads it about the hacking scene in the 1990s, as well as how one of the most accomplished hackers of that era went about learning his craft.

We also have reports from the Security Symposium and many of the co-located workshops. We had a good crew of summarizers, as well as lots of fascinating presentations. I particularly enjoyed the Symposium, which I spent mainly in the Papers track, as well as WOOT and HotSec. But those just represent my own interests.

Besides the summaries, USENIX also provides both recordings and videos of most presentations. These are available online, via the pages for the Symposium and the workshops. Also, as of this August, thanks to the support of you, the members, USENIX expanded its Open Access policy to open all videos to everyone.

I want to leave you with a couple of thoughts. Back in the '90s, when Kevin Mitnick was plying his skills with great success, few people knew, or cared, about computer and network security. That lack of concern helped people like Mitnick simply because people were not expecting to be compromised.

Today, things are a little different. Having watched organizations that should *not* be vulnerable fall to fairly simple attacks, organizations such as security companies, security contractors, defense contractors, and even a certificate authority, expectations are different. People have grown to expect that their systems will be exploited. That is even more true for desktop users, where being able to do whatever the user wants to do, including entertainment, is the norm for both work and home systems.

If we want to have more secure systems, we will need to accept some changes. The Sassaman proposal that appears in this issue provides a concrete step toward creating more secure programs in the future, as well as identifying programs and protocols that can never be made secure. Perhaps the best we can do is to sandbox the unsafe programs as best we can. But that works poorly when the program in question has access to our personal information or defense secrets. In those cases, the Sassaman proposal becomes a much more critical tool for deciding what protocols can be used when security is required.

References

[1] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley, "Q: Exploit Hardening Made Easy," *Proceedings of the 20th USENIX Security Symposium (USENIX Security '11)*: http://www.usenix.org/events/sec/tech/full_papers/Schwartz.pdf.

[2] James Oakley and Sergey Bratus, "Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code," 5th USENIX Workshop on Offensive Technologies: http://www.usenix.org/events/woot11/tech/final_files/Oakley.pdf.

[3] The Firewall Toolkit: http://www.fwtk.org/fwtk/.

[4] Dutch CA compromised: https://www.net-security.org/secworld.php?id=11565.