Musings

RIK FARROW

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

I've sometimes been asked why computers are still so insecure, so eminently hackable. Didn't Bill Gates once shut down development at Microsoft so they could improve the security of Windows decades ago? While not quite decades ago, Gates really did shut down Windows development in 2002 and sent 7,000 systems programmers to special security training with the goal of "Trustworthy Computing."

It didn't work. While some things got better, and the rampage of worms slowed down, administrators and users continued to have to install patches frequently. In 2003, Patch Tuesday became a regular feature, followed by Exploit Wednesday for those who had ignored the routine of installing patches on the second Tuesday of the month.

Part of Microsoft's problem was a matter of programming culture, with a focus on new features. Exchange server, the email server product, actually had a good record for security, while the IIS web server certainly did not. Two distinct groups, with a different culture, worked on these products, resulting in very different security outcomes.

But the problem of insecurity is not unique to Microsoft. Sun Microsystems started delivering insecure workstations in the early 1980s, and continued to do so through the '90s. Sun employees announced at USENIX Security that they had a program for securing SunOS, but it was for internal use only. Dan Farmer, Brad Powell, and Matt Archibald released Titan for Solaris in 1998 as a public solution to tightening and securing Solaris. Linux was having severe issues with security at the end of the '90s but quickly improved over the next couple of years. But today, people build malware specifically for Linux, as Linux servers and desktops have become important targets for invading networks.

So far, all I've done is write about how the struggle to defend software against exploits has been a failure, but not why. The answer lies partially in the nature of software and largely because of our hardware designs.

First, programming is hard. I am constantly amazed at people announcing that they intend to turn everyone into a programmer. Perhaps these well-meaning projects can turn some people into middling programmers, but not ones who will be writing the next generation of services. I have had the misfortune of consulting in IT shops and have seen the carnage firsthand. On the plus side, when I turned out a handful of lines of shell script that did what they had failed to do in weeks, it made me look like a wizard. I have said this before: most programmers, by definition, have an average skill level, and half are below average. This is hard to remember when you work in Silicon Valley or at a top-ten university and all of your coworkers are geniuses.

Second, our computer systems were not designed for security. They were designed to be flexible. There are hardware security mechanisms that are important to security, such as the so-called rings, with the lowest numbered ring having the most access to hardware, and higher rings being reserved for "untrusted" code. Yet the largest and most complex programs run on most systems are the operating systems, and these run at the innermost ring. That makes the operating system the most important target for any attacker.

Microsoft has taken advantage of the ring added to support virtualization, called ADM-V or Intel VT, in Windows 10. They load kernel modules using Virtual Secure Mode, where the operating system and critical system modules get executed in virtual containers. This beats the pants off the Linux model, where the kernel resides in a single address space, but still hasn't prevented bootkits from being installed in Windows 10 systems. This is supposed to be prevented by UEFI, but this can be worked around using firmware rootkits and on many motherboards because of the wrong settings being used.

Memory management is the next level of protection, but it was designed to protect programs running in one process from programs running in another process. Through abuse of the operating system, usually after an exploit, memory management can be bypassed.

Intel has introduced another level of protection, although this one is largely unused today. MPK (memory protection keys) allows programmers to split a single process's memory space into 16 different regions with the same protection provided by page tables [2]. Sixteen regions doesn't sound like a lot, but as a method for isolating threads, or portions of a program involved in parsing input, MPK could help.

The CHERI researchers have taken a slightly different tack by creating CPU designs with segment registers. MULTICS used segment registers to separate portions of programs, with a segment having a base address and a range, and accesses outside of this base and range being prohibited. CHERI represents another great idea, one that's been in development over a decade, making segments associated with capabilities, and one quite unlikely to be adopted by most programmers.

I guess I should mention enclaves, the tiny, encrypted execution domains, so I can also mention Meltdown, Spectre, and Load Value Injection [1]. Enclaves will not be of use to most programmers, and transient execution flaws have painted targets on them already.

Software

That leaves us with software. Software can either make computers more secure or less secure, and our favorite languages make our systems less secure.

```
.cfi_startproc
pushq   %rax
.cfi_def_cfa_offset 16
movslq  %edi, %rax
leaq    _ZN5hello4main17hd078db076938ab99E(%rip), %rdi
movq    %rsi, (%rsp)
movq    %rax, %rsi
movq    (%rsp), %rdx
callq   _ZN3std2rt10lang_start17he5a718dea3bb834eE
popq    %rcx
.cfi_def_cfa_offset 8
retq
```

Listing 1: Some assembler

Listing 1 depicts the `main()` function for a "Hello World!" program. Compilers produce assembler as an intermediate format, and that's what appears in Listing 1. You can learn to program in assembler, but you have to handle things that compilers make easy to do, like choosing the register to use (anything beginning with `%`), managing the stack, managing memory. Each CPU architecture has a different set of registers and assembly instructions, although assemblers themselves, like `as`, work the same. You still have comments, but assembler is hard to read and is not portable between CPU architectures.

That's why the geniuses who created UNIX created the C language: they needed a language that made porting an operating system easier. They also wanted something that would be fast and that provides little in the way of handholding. If you don't know better, you can easily make "fatal" mistakes, like using a pointer after the memory it points to has been freed or writing into memory beyond the end of an array. On the other hand, you can treat pointers as function entry points and perform arithmetic on pointers, very handy things to have when writing an operating system—especially one that runs on hardware with 32K words of RAM.

C is my favorite language, but it is a language without seatbelts, airbags, or even bumpers. C, and its younger cousin C++, assume that you know what you are doing and you never make a mistake. The first of these points is rarely true, and the second is never true—even the best programmers make mistakes.

There are safer languages to use, ones with safety features. Generally, these languages remove access to pointers and provide strong types. Go and Rust are examples of safer languages, with Rust being designed particularly for safety. Go is not as fast as C or C++, but perhaps a 10–15% penalty for a lot of execution safety is worthwhile. Rust, meanwhile is nearly as fast as C, and perhaps will be when LLVM can produce code as performant as GCC.

Safer languages leverage hardware support for security by making it much more difficult to write programs that are terribly insecure. I think this is a very good idea, especially if we are going to teach everyone to program.

The Lineup

We start out this issue with an opinion piece by Michael Mattioli, who feels that tools like Zoom, Meet, Teams, and so on are missing something important.

Next, I picked two papers from USENIX Security '20 that were clearly written and included points that I felt were especially worth sharing. There were another half-dozen papers that I really liked, but those either weren't as well written, had deep dives into statistics, or were too narrow for the wide audience represented by the USENIX membership.

Votipka et al. examine programmer mistakes, but not just any type of errors. They used the Build It, Break It, Fix It (BIBIFI) program as their data source. BIBIFI challenges programmers who have had training or work experience to write three, non-trivial programs with some security requirements, share the sources to these programs with other teams, and then analyze the programs and the faults found by the teams. What they found was distressing to me and is part of the reason why security is so hard to get right.

Garfinkel et al. have written a tool, RLBox, that makes sandboxing libraries easier. Most programs incorporate libraries, and many of these libraries process input that may come from attackers, such as image or video decoders. RLBox simplifies the process of sandboxing these libraries. The authors worked with Mozilla to sandbox several key libraries, and their tool will work for other programs as well.

Huang et al. volunteered to write about their research project, Senx, an automatic program repair tool. The authors argue that waiting for security patches to appear often takes much too long, and with access to source code and an example of an attack, Senx can create patches for three different types of vulnerabilities.

I interviewed Sergey Bratus. There were several papers at USENIX Security '20 that appeared to be directly related to Language Security principles, or LangSec. Bratus has written for *;login:* before, has been running a workshop on LangSec for years, and seemed to me to be the perfect person to explain LangSec principles. And this worked, as LangSec seems much clearer to me now and is important if we are ever going to be able to write secure software.

The USENIX Annual Technical Conference also happened this summer, and I chose two papers and one talk as the basis for articles. Shahrad et al. explain a key feature of running cloud functions: deciding how long a function should be kept warm, that is, ready to run. They provide examples taken from Azure and a new scheme that improves performance and efficiency.

Raghavan et al. discuss Posh, a distributed shell. To me, Posh is a great example in the tradition of USENIX ATC, an improvement on the shell that works by moving execution closer to the sources of data, when that data is available over NFS. Posh can also add parallelism to shell scripts without rewriting the scripts.

I interviewed Margo Seltzer, who gave an afternoon keynote at USENIX ATC '20. Seltzer encouraged her audience to explore beyond the safe confines of their personal specialties and consider "fringe" ideas. Seltzer provides several examples of doing this in her own highly successful career.

Torres and Colish cover capacity planning for SREs. They divide capacity planning into two areas: resource provisioning and capacity planning to safeguard the future potential of a service. The authors cover redundancy for reliability and how this must include back-end services as well.

Adam McKaig explains why he thinks that it's important for SREs to understand algorithms and data structures. McKaig takes us through three examples of a service that initially is performing well, uncovering the reasons why the service starts failing SLOs, and explaining the solutions that he and the teams he worked with came up with for repairing the service.

Laura Nolan has written a book review of Alex Hidalgo's recently published book about SLOs. Nolan explains why she considers Hidalgo's book one of the most important for SREs. Hidalgo wrote an article for *;login:* in the Summer 2020 issue, so you can also sample his writing there.

Cory Lueninghoener shows us how to create different aspects of containers from the command line. While you may be more likely to use a tool like Docker for this, you will gain understanding of what Docker is doing by trying Lueninghoener's examples.

Dave Josephsen continues his exploration of eBPF, this time focusing on histograms as a clever technique for displaying potential performance issues. Josephsen dives into how to select bin sizes for histograms and exactly why histograms are so good at unveiling problems that would be buried in data otherwise.

Simson Garfinkel covers the history and uses of cryptographic hashes. While the use of hashes has become commonplace in programming, cryptographic hashes provide the foundation for assuring the authenticity of code or messages, timestamps for documents, and in forensics.

Terence Kelly demonstrates a technique for storing graphs as compressed sparse row format. First, Kelly shows the most commonly used ways of storing graphs, explains why these methods waste memory, and then details how to use the compressed sparse row format and when other formats will work better.

Dan Geer, along with coworkers John Speed Meyers and Bentz Tozer, has researched software supply chain insecurity. They have collected data about how often attackers have modified the source code for open source libraries as well as how often this has resulted in successful attacks, work that I believe is really important so long as we continue to include other people's code, via libraries, in our own programs.

Robert Ferrell distracts us with his views on social media *influencers*. Ferrell deletes himself from this clan, while pondering on the usefulness of content that is itself nothing more than advertising.

Mark Lamourine has reviewed three books this time, *Effective Python, Dependency Injections, Practices, and Patterns,* and *Building Secure and Reliable Systems*. I reviewed a book about rootkits for Windows.

Most of us have little to no influence on hardware design. To be honest, most of us won't have the type of ideas necessary to even get the CPU industry to move at all toward better security. Personally, I'd like to see designs that support message passing without involving context switches, as that would allow our servers to appear more like clouds than 1970s mainframes.

We do have choices we can make about the programming languages we use. Well, some of us do, while those working at corporations often have that decision made by someone far off in the top of the management hierarchy, based on the latest buzz. For those who have choices, I recommend languages like Rust that emphasize both security and performance. For a different way of looking at things, I found this article at Northeastern an interesting way to view programming languages [3]. Hopefully, someone will keep this page up-to-date so we don't have to rely on sites like TIOBE.

And as for making systems more secure, we do need to stop handing out assault rifles like C++ and get more people to use inherently safer programming languages like Rust or Go. Python has its faults, like the lack of strong types and being single threaded like JavaScript, but it doesn't have pointers and the types of memory issues that C and C++ have had for decades. Decisions at institutions of higher education do have an influence over the future security, or insecurity, of computers.

Remember Listing 1, the assembly language example? That was `helloworld.rs`, the Rust version, but you can hardly tell by looking at the intermediate assembly code. All programming languages wind up as machine code, and while that may sound like all languages are equal, they are not. Some languages take advantage of advances in compiler designs so they make it much easier to write secure code. You can choose the 1970s model with some upgrades, or learn something new that can help make the world a safer place.

The Future of *;login:*

This issue marks the end of print *;login:*. You can read about why this is happening and learn more about how the digital version of *;login:* will work in *USENIX Notes*, beginning on page 95.

Some people have found the reference to "peer-reviewed" in this description a bit confusing, thinking that the digital version of *;login:* will be like a journal. That's not true. The peer-review has long been a part of editing *;login:*, and consisted of PC members who accepted the papers that many articles are based upon. For the rest, I was the "peer," with responsibility for accepting articles only from subject matter experts. I did rely on other experts in areas where I was unfamiliar with the authors. The digital version will expand the number of peers, so I will no longer be responsible for culling out articles that should not be published in *;login:*.

Another advantage of a digital *;login:* will be shorter elapsed time between submitting an article and its appearance online. Printing *;login:* takes a long time—just dealing with the printing process itself took almost three weeks. While I might see a draft, get a final version, format it and turn it in in just one week, the process that includes copy editing and typesetting takes a great deal longer. Michele Nelson, the Managing Editor, received articles from me and sheparded them through this long process.

I think we will miss our copy editors, Steve Gilmartin and Amber Ankerholz. Good copy editors improve your writing, often taking something not written that well and turning it into something that makes you start believing you really can write well. The copy editor must improve your written English without distorting your meaning, and Steve did a great job. Amber's task was to approve Steve's edits from a technical standpoint. That process, and proofreading, added three weeks to the process. Typesetting, expertly done by Linda Davis, added yet another week. When you add all of this up, and start from the point when I ask authors to write or get a proposal, the process can take over four months. I don't even want to think about how long your *;login:* magazine sat in a pile before you started reading it....

The digital *;login:* will be open access. Laura Nolan has written about the value of open access in this issue. All articles will be open access when posted, as opposed to members-only for one year. Only USENIX members will be able to comment on articles, something we hope will lead to discussion about articles and feedback to authors. With print *;login:*, about the only time authors get feedback is during in-person conferences, and from personal experience I can tell you that even that is rare. I hope the ability to respond to articles results in useful feedback, or at least acknowledgement that someone has read and appreciates the work someone put into an article.

We—that is the committee composed of three board members, Laura Nolan, Arvind Krishnamurthy, and Cat Allman, along with Casey Henderson—came up with several other ideas to celebrate this, the final print issue. I was assigned to interview two early USENIX members. Clem Cole has the honor of being USENIX member number seven, and I interviewed him first. Kirk McKusick represents, at least for me, the Berkeley side of UNIX and makes up the second interview. They both participated in the story of how USENIX helped Rick Adams start UUNET in the late 80s, as did Deborah Scherrer (the Board VP), Steve Johnson (Treasurer), and Rick Adams. Adams recommended reading Peter Salus' (Executive Director) article [4]. Adams also deserves thanks for donating UUNET stock from his fledgling company that later became the foundation for the endowment that is keeping USENIX alive during COVID-19.

Finally, Laura Nolan, Arvind Krishnamurthy and I picked out our favorite articles from *;login:* issues starting with 2005. I learned that my ability to edit *;login:* has improved over the years. I had started to edit special security-focused issues of *;login:* in 1998, but to my eyes, the first five years of being the regular edi-tor, starting in 2005, seem pretty rough.

You might be wondering what I plan to do with all the time I will have because I will be sharing the editorial responsiblities. I plan on writing some science fiction, and hope to have at least one short story up at https:/rikfarrow.com/fiction/ by the time this issue appears. I've started at least five stories, and have one close to completion—about computers and future myths, of course.

References

[1] D. Goodin, "Intel SGX Is Vulnerable to an Unfixable Flaw That Can Steal Crypto Keys and More," Ars Technica, March 10, 2020: https://arstechnica.com/information-technology/2020/03/hackers-can-steal-secret-data-stored-in-intels-sgx-secure-enclave/.

[2] Memory Protection Keys: https://www.kernel.org/doc/html/latest/core-api/protection-keys.html.

[3] B. Eastwood, "The 10 Most Popular Programming Languages to Learn in 2020," Northeastern University Graduate Programs, June 18, 2020: https://www.northeastern.edu/graduate/blog/most-popular-programming-languages/.

[4] P. Salus, "Distributing the News: UUCP to UUNET," *;login:*, vol. 40, no. 4 (August 2015): https://www.usenix.org/system/files/login/articles/login_aug15_09_salus.pdf.