Interview with Peter Gutmann

RIK FARROW

Peter Gutmann is a researcher in the Department of Computer Science at the University of Auckland working on design and analysis of cryptographic security architectures and security usability. He helped write the popular PGP encryption package, has authored a number of papers and RFCs on security and encryption, and is the author of the open source cryptlib security toolkit, *Cryptographic Security Architecture: Design and Verification* (Springer, 2003), and an upcoming book on security engineering. In his spare time he pokes holes in whatever security systems and mechanisms catch his attention and grumbles about the lack of consideration of human factors in designing security systems. pgut001@cs.auckland.ac.nz

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

I probably first met Peter Gutmann at a USENIX Security conference. I really don't remember which one, but Peter was also a friend of an acquaintance, which led to the usual social lunches during conferences.

Peter seemed like an "odd bird," and he's actually a Kiwi, a person from New Zealand, making that part true enough. But I always enjoyed talking with him.

I recently had read some of his postings about the failure of a very prominent crypto library (think Heartbleed) and thought now would be a good time for an interview, one that would allow me to ask some questions about programming and cryptography, and get answers that I felt most people would understand.

During the interview with Peter, conducted over email, the person I had asked to write about the AFL fuzzing tool dropped out, and Peter mentioned that he had given a talk about AFL at a Kiwi-con and was willing to write about AFL as well. As if this wasn't enough, I had asked Peter to review a book about iOS security well before any of this started (it takes months to get publishers to ship books to New Zealand), so Peter has almost as many pieces in this issue as I do.

Rik: Your first USENIX paper [1] had to do with remanence: that data written to hard drives and later erased or overwritten could often still be recovered. I think you were working for IBM at the time, and I imagine you had access to some unusual/sophisticated hardware in order to image partial tracks on disk drive platters.

Peter: Actually that was written some years before I was at IBM. I communicated via email with a few people who had worked in the area for general details on what was involved in reading disk tracks with MFMs and stories about deletion (and lack thereof) and heard a few interesting stories, but that sort of thing would have required access to pretty specialized hardware to do. By "specialized" I mean "not in your standard lab" but readily available to hard-drive manufacturers. That's how the sampling-scope drive read that I mention in the paper was done, something that was revived in 2011 to recover data off an old Cray-1 disk pack [2].

Rik: Did you work in New York for IBM? What did you do there?

Peter: I was at Watson Labs in Hawthorne (OK, "the IBM Thomas J. Watson Research Center," an offshoot of the main one in Yorktown Heights) as a visiting scientist working on my thesis. The idea is that IBM gets people in from all over to work there for a while with the hope that eventually they decide to stay. It was a fantastic place to work; if you ever had a question about something, there was a good chance that an expert on the topic was just a few doors down the hallway. Not to mention access to their extensive library, and people who had been in the industry for years (or decades). This was mostly before archives of paper journals and conference proceedings were scanned and put online (and some were never put online), so a lot of the background material in the thesis was gathered from there.

Rik: In other work, you programmed a library called cryptlib [3]. Why did you decide to create your own cryptographic library when others were available at the time?

Peter: cryptlib was from 1995, when what was around was mostly Eric Young's libdes. The underlying code goes back even further, to about 1992 or 1993 in the HP ACK archiver (I should have trademarked that name, given its reuse in HTTP 2), which offered digital signatures and public-key encryption of archives and assorted other things, and that was based on work on PGP 2 from even earlier.

Initially it was just something I did for fun, but then in about 1997 I was persuaded to license it commercially. Before that I'd refused to take money for it, which caused problems with some users because they needed a commercially supported product and not just open-source throw-it-over-the-wall. So it's dual-licensed under the Sleepycat license: you can use it as open source or commercially supported if you need that.

Rik: So you have firsthand experience with supporting a software package that has a dual open source/commercial license. Not many people do that on their own. What are the pros and cons?

Peter: There are pros and cons to being OSS and being commercial. OSS is obviously free, but it's also often throw-it-over-a-wall stuff; if you have a problem with it then you google Stack Overflow or post to a mailing list and hope that at some point someone volunteers to help you. Lots of commercial organizations can't work that way; for starters, since their work is security-related you need an NDA. Then you need commercial support with a guaranteed response time. If [the organization] has a problem, they want to get someone on the phone or on-site who can walk them through dealing with it; they want a standard commercial license (which often boils down to them knowing that there's someone they can sue if things go wrong); and so on. In broader terms, they need to deal with risk management: is there a commercial entity behind this? will they still be around in a year's time? will it be supported in 10 years' time? and again, in general, what do we do when things go wrong?

Having a commercial option has been really useful. With open source you do essentially just throw the code over the wall and sometimes you get feedback, but mostly you don't, so there's little opportunity to enhance the code based on users' needs because if someone has a problem you rarely hear about it; they either patch in a "fix" themselves or go elsewhere. Commercial users, since they have a guarantee of support, will come to you with issues, and so you can then use that to improve the code and work with them to solve the problem. Virtually all commercial users have agreed to having specific fixes put into the main, supported code base, because the last thing they want is to have to deal with a custom version for the rest of eternity. That's kind of weird really; in the OSS world, everyone seems to be happy to fork off their own special-snowflake version with their own code in it, while the commercial users want a single, stable, supported code base and are OK with sharing the code changes.

The much longer answer can be found at [4].

Rik: The Internet of Things, IoT, looks like it will include anything with a computer that is not a personal computer or device, or a server, and that is connected to the Internet. To my mind, that poses certain very real threats: to the security of the device itself and to the security of the data the device collects and transmits. You have been working with cryptography for over 20 years. Do you think that people can develop reasonable ways to handle cryptography on these low-powered devices? We still haven't solved key management on the larger devices we are already using.

Peter: We haven't even solved basic crypto on these devices. This is a bit of a pet peeve of mine. It's scary the number of times I've seen people on security mailing lists announce that in the future we'll all have infinite CPU and RAM and therefore can design arbitrarily baroque and complicated protocols and don't have to worry about resource constraints. Only last week someone pointed out that the just-appeared Raspberry Pi 3 has *X* resources and so we don't have to worry about resource-constrained embedded anymore. A Raspberry Pi of any kind, including the Pi Zero, isn't an embedded device, it's a PC. So is any smartphone, tablet, router, WiFi access point, and a long list of other items. Citing Moore's Law won't help because a significant portion of the market uses it to make things cheaper rather than faster, so that performance stays constant while cost goes down rather than the usual PC equilibrium of cost staying constant while performance goes up.

An embedded device is something like a Cortex M3, or more recently the M0, a 32-bit CPU perhaps clocked at a blazing 40 or even 70 MHz, with something like 256-kB flash and 32-kB RAM. Ten years ago the state of the art was a Cortex M3, while today it's a cheaper Cortex M3, and in 10 years' time it'll still be a cheaper M3 (or possibly an M0+++ by then). No standard security protocol will run on that. A week ago I got to review two proposed ISO standards for IoT in which a bunch of networking engineers tried to invent some sort of crypto mechanism that makes WEP look like a model of good design, because the obvious candidates TLS and SSH are far too bloated to work for them. It's not their fault; they're networking engineers and shouldn't be expected to have to do this, but the crypto community just assumes infinite resources and goes from there. So we've got a desperate need to secure IoT but no widely accepted standardized protocol that works for it.

This is already a problem with smart meters because regulators imposed requirements for certificate-based signed messaging and updates onto CPUs like TI MSP430s, Motorola ColdFires, and ARM Cortex-Ms, and some clocked as high as 16 MHz and with as much as 32 kB of RAM (for everything, not just the crypto). The solution with smart meters was to cut corners as much as possible in order to make things fit, skipping certificate

verification, assuming hardcoded public keys, and various other measures that are destined to become entertaining Black Hat or DEFCON presentations in the future.

That's a really short version of what could turn into a really long answer. I haven't even touched on the problem of dealing with the fact that these devices will need to work in rough environments where the hardware will experience faults, while the fashion seems to be to move towards extremely fault-prone algorithms like ECC crypto (where almost any kind of fault tends to result in the private key being leaked) and GCM-mode encryption, where a single fault, failing to increment the counter value or change the IV, results in a total, catastrophic loss of security. So that's been another ongoing project: since embedded devices are going to experience faults, how resistant can you make your crypto to random (or perhaps deliberate, maliciously induced) faults?

Rik: I noticed that cryptlib is written in C. Why not use a "safe" language, one with built-in safeguards against exploitation?

Peter: Whether C is safe or not depends on what you mean by "safe." In many high-assurance/safety-critical applications, C is regarded as safe, and languages like C++ and Java are unsafe, because with C you can establish pretty clear correspondence between the C code and the resulting binary, while with higher-level languages you can never really be sure what's going to happen.

The FAA, for example, one organization that really cares about safety, has spent about 10 years trying to develop guidelines for the safe use of OO languages (typically C++ as a follow-on from C), but is still having problems dealing with dynamic dispatch, multiple inheritance, polymorphism, overloading and method resolution, and other aspects of OO programming (see the DO-332 supplement to the DO-178C avionics software standard for more on this).

When you're operating in environments where you can't have recursion (you could run out of stack), you can't have dynamic memory allocation (it leads to nondeterministic behavior in the program), and you need to perform worst-case execution time (WCET) analysis on every routine to make sure it doesn't block, or stall, time-critical code; the less fancy high-level stuff your language has, the better.

Another thing about C is that the language is simple enough to have a huge amount of tool support available. I was recently re-reading Les Hatton's *Safer C*, a seminal book [5] from the mid-'90s, and even then he was comparing criticisms of C that were mostly based on K&R with the then-current analysis tools that went way beyond what the standard said in terms of code checking.

Things haven't stood still since then. You've now got incredibly sophisticated tools like Microsoft's PREfast that can treat C almost like Pascal or Ada. For example, they'll tell you that a variable that you've said has the range 0...1000 has been assigned a value of 1001 (or whatever). That's something that looks like C on reading, but which can be analyzed by the dev tools as if it had Pascal or Ada's type-checking.

The reasoning that some of these tools can apply is phenomenal. The clang analyzer, part of the LLVM compiler suite, can do things like tell you that if you enter this loop and take these code branches, then after going through five iterations this unexpected result (e.g., a pointer value being null) will occur (commercial tools like Coverity do this too, but in even greater detail). That's something that no human would be able to detect, and that testing probably wouldn't ever turn up either because you may need to go through 20 or 30 distinct steps to get to that point.

So a programming language is more than just something to translate into object code, it's the sum of the tools available that support it. Consider, for example, the use of the AFL fuzzer that I talk about in the AFL article on page 11 in this issue. That uses compiler-based instrumentation to detect memory issues with the address sanitizer ASAN, not just out-of-bounds accesses but other problems, like use of uninitialized memory and so on, and more instrumentation to do execution-path analysis to maximize code coverage by the fuzzer. Now imagine trying to do that with a JVM, where something outside your control (the virtual machine) is dealing with most of the stuff that's exposed in C. How would you fuzz that? You end up either missing a lot of the stuff that needs to be fuzzed, or fuzzing the JVM itself rather than the program you want to check.

References

[1] Peter Gutmann, "Secure Deletion of Data from Magnetic and Solid-State Memory," in *Proceedings of the Sixth USENIX Security Symposium,* 1996: https://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html.

[2] Recovering data from an ancient disk pack: http://www.chrisfenton.com/cray-1-digital-archeology/.

[3] Cryptlib: https://www.cs.auckland.ac.nz/~pgut001/cryptlib/.

[4] "Self-Sustaining Open Source Software Development," Conference for Unix, Linux and Open Source Professionals (AUUG2005), slide deck: https://www.cs.auckland.ac.nz/~pgut001/pubs/oss_development.pdf.

[5] Les Hatton, *Safer C: Developing Software for High-Integrity and Safety-Critical Systems* (McGraw-Hill International Series in Software Engineering, 1995).

[6] Peter Gutmann's home page, with lots more pointers to slide decks and other materials: https://www.cs.auckland.ac.nz/~pgut001/.