Writing Secure Programs

An Interview with Steve Lipner
by Peter J. Denning, Ubiquity

Editor's Introduction

Protecting computing systems and networks from attackers and data theft is an enormously complicated problem. The individual operating systems are complex (typically more than 40 million lines of code), they are connected to an enormous Internet (on order of 1 billion hosts), and the whole network is heavily populated (more than 2.3 billion users). Hunting down and patching vulnerabilities is a losing game.

Steve Lipner, partner director of program management in Trustworthy Computing Security at Microsoft, has been involved in securing systems for nearly 40 years and has learned how to make security better. His responsibilities encompass Microsoft's process for assuring the security of its products and online services— the Security Development Lifecycle (SDL)—as well as a variety of programs related to government evaluations of the security and integrity of Microsoft products and services. Lipner has been a consultant, researcher, development manager, and corporate executive in what we refer to today as "cyber security." Here he shares his experiences in what has and has not worked. He sees by far the best results when programmers adopt secure development practices.

Peter J. Denning
Editor-in-Chief

Writing Secure Programs

An Interview with Steve Lipner
by Peter J. Denning, Ubiquity

Ubiquity: Everyone seems to be concerned about cyber security these days. You've been involved with that for a long time. Please tell us how you got involved in this area and why.

Steve Lipner: In the early 1970s, I was working on Air Force contracts for the MITRE Corporation and had just completed an assignment on site at an Air Force base. On my return to MITRE headquarters outside Boston, my boss asked me to get involved in a new project— multilevel security and the development of computer systems that could protect classified information. I was much more knowledgeable about application programming than operating systems, and told him that I thought he should find someone with a real degree in computer science, but he said there was no one with that background available and asked me to fill in until we could hire the right person. Of course, MITRE never did end up hiring that right person to replace me at the time and I've stayed in the field ever since.

Ubiquity: You were involved with early attempts in the 1980s, notably the so-called "Orange Book," to certify systems as secure. What did it mean to be secure? Did anyone ever produce a fully secure system and get it certified as such?

SL: In the 1970s and early 1980s, when we were doing the research that preceded the Orange Book, and then while the Orange Book was the dominant security evaluation policy— from 1983 through the early 1990s—"secure" meant the ability to protect labeled, classified information in a computer system from being released by an intruder or a Trojan horse program acting on behalf of an intruder. There were a variety of technical requirements to certify security, including formal verification of system design and testing for "covert channels."

Ubiquity: So the Orange Book was an attempt to automate security, modeled after military security and some corporate security need-to-know systems?

SL: Yes. The Orange Book was strongly based on information security policies for national security where information has classifications and users have clearances. You can see

information only if your clearance is as high as or higher than the classification of the information.

Ubiquity: What does "labeled" mean?

SL: The system marks each user and each item of information with a tag. The user tags represent clearances and the object tags represent classifications. The tags themselves are ordered so that the system can quickly check whether one is higher, equal, or lower than another. This makes it easy for the system to check each operation for security, before it is performed, simply by comparing tags. Tag-checking implements the notion that I cannot read any information for which I am not cleared, and I cannot write any information to a file or object tagged lower than my clearance. It's a clever scheme that handles traditional security levels such as unclassified, secret, and top secret. It also handles compartments.

Ubiquity: Did you ever try to build one yourself?

SL: My team at Digital Equipment attempted to build a secure system that met Class A1 (the highest) of the Orange Book. It turned out to be much harder than anyone thought! There are so many ways to leak information and you have to prove every one of them has been closed off. Eventually, I had grave doubts whether our system could be commercially viable. I had to cancel the project. But I like to think we came close. A couple of other systems actually achieved A1 certification, but none was a commercial market success. Cancelling our project was the right thing for our business.

Ubiquity: If A1 systems were too hard to build and certify, was the whole point of the Orange Book specification of security a waste? Did anything good come of that?

SL: That's not an easy question. The Orange Book did establish the practice of third-party evaluation of commercial products using government standards. That practice continues to this day under the international Common Criteria. Moreover, some of the high assurance concepts and models have benefited from high-security systems used by government. But in the 1980s, we believed the high assurance concepts would gain adoption in the commercial market, and that belief was just wrong.

Ubiquity: One of the attacks I hear about a lot is the "buffer overrun." What is that?

SL: Buffer overruns are a class of programming error where a program does not check that an incoming argument string will fit into the internal buffer space allocated for the string. Here's how it works. When a program calls a procedure, the operating system allocates a new block of storage for the called procedure. That block has a fixed amount of space for the procedure's parameters. If the caller provides a parameter bigger than the space allocated inside the procedure block, the extra bits overflow into other memory locations such as the one holding the return address. An attacker can exploit this by providing an oversize parameter with another address at the precise position that will overwrite the return address. This will cause the procedure to return to the wrong place, which could actually be the entry point of a Trojan Horse.

The remedy is to set up the procedure or operating system to check that each incoming argument string fits in space allocated for it. Many programmers omit such checks, even after being told to put them in, because it's very easy to overlook every scenario where a length needs to be checked. Buffer overruns are pretty intimately tied up with the C and C++ languages. Those of us with long memories are aware of the problem being exploited in assembly language procedures on mainframes from the 1970s. The Internet Worm of 1988 exploited a buffer overrun. However, the general knowledge of buffer overruns and how to exploit them wasn't widespread until the 1990s.

Ubiquity: Some operating systems avoid this problem by storing indexable objects like arrays off the stack, in a separate area called "heap." Does that solve it?

SL: Not really. The overflow won't mess up the call stack, but it can mess up the heap. The vulnerability research community has studied heaps as well. We've offered three kinds of guidance.

First, guidance for developers to try to keep them from making errors. For example, "Check your array and string lengths."

Second, methods to block errors that slip through. For example, checking the integrity of key stack elements before returning and perhaps using a corrupted return address.

Third, mandates for "mitigations" at compile, link, or build time so that any attempts to exploit vulnerabilities will cause runtime errors instead of executing malicious code.

Ubiquity: Could the Orange Book techniques prevent that vulnerability?

SL: That's a good question. Some Unix systems coded in C underwent Orange Book evaluation (though not at the highest levels) and I'm pretty sure the evaluations did not discover or remove buffer overruns that were surely present in those systems. When we built the A1 system at DEC, we chose Pascal and PL/1 as "off the shelf" languages that were suitable for "structured programming"; neither of those languages is susceptible to stack-based buffer overruns as C and C++ are. Other classes of buffer overruns have been discovered over the years (heap overruns, for example) but I don't think anyone has ever gone back to see how those affect some of the older languages.

Buffer overruns illustrate an important aspect of our evolving knowledge of security. Over the last 20 years (or more) security research has discovered new classes of attack that no one had previously been aware of. This research has changed the rules for what being secure means and required us to take new measures in the process of building secure systems. The Orange Book had nothing to say about either buffer overruns or many other vulnerabilities. Orange Book evaluators probably looked for buffer overruns after they were discovered. The Orange Book itself did not specify how to write a secure system—it told you to counter the threats that were understood at time the Orange Book was written.

Ubiquity: You mentioned covert channels. What are those? Why are they hard to block? Does anyone actually use them?

SL: Covert channels are ways that a malicious program—a Trojan Horse—can communicate information from a hijacked process running on behalf of an authorized user to a process running on behalf of an attacker. The Trojan Horse can do that by encoding pilfered information into a storage location that's not subject to the system's security policy—an example might be a file's length or "last used" date—or by encoding information into the running time of the Trojan horse. We call these storage and timing channels, respectively. An example of a timing channel might be consuming lots of CPU time to signal a "1" and waiting (consuming no CPU time) to signal a "0."

Covert storage channels are pretty easy to block in a system that enforces labeled protection on all objects—we just label every variable that any process can access. Covert timing channels are devilishly hard to detect and block. We tried to block or limit timing channels in the A1 system and were only partially successful.

We see lots of Trojan Horse attacks although most are not exploiting covert channels. We've also seen descriptions of covert timing channels against cryptographic software, although again it's unclear whether channels like that are actually exploited. Although covert channels are a theoretical threat, I don't see a lot of evidence that attackers use them very much.

Ubiquity: What has changed since the 1980s that makes security harder today?

SL: The Orange Book did not anticipate networked systems or systems as complex and feature-rich as we have today. For example, our original A1-hoped-for system was a time-sharing system with dumb terminals; but before the system was complete, users had migrated to powerful workstations or personal computers with rich graphical user interfaces. It would have been a major research project to figure out how to build a highly secure system with that kind of user interface. And we only had the haziest idea of how to interconnect secure systems of the sort we were building— another feature that users came to expect and would have required a major research project to develop.

Thus, by the late 1980s, our experience seeking evaluation at the higher levels of the Orange Book seemed to be an open-ended research project as we and the evaluation teams explored properties of those systems that no one had considered in depth. Freedom from covert channels is a great example—my team developed a pretty cool approach to finding and removing timing channels. We published our method at a research conference in 1991. In 1992 a Navy researcher published a paper that told how to defeat our cool approach. By that time our project had been cancelled. Had it continued, we surely would have undertaken more research to defeat the Navy researcher's counterattack. Do you see the never-ending cat-and-mouse game here?

Security today is hard because we are working on complex systems that dwarf the primitive systems of the 1980s. Attackers have discovered techniques that we did not know or understand in the 1980s. Security is an unending competition between attacker and defender. There is no simple set of requirements that say "do this and you're done."

This is even true of that old nemesis, buffer overruns. We have developed good tools and countermeasures to block simple stack overruns in C and C++. More recently researchers have discovered attacks on the arithmetic methods that calculate string lengths and the allocation of heap memory. We have fewer defenses against these kinds of attacks, and the defenses we do have are more difficult for developers to implement correctly. We're continually working to improve this situation, but attackers are also continually working to invent new classes of

attacks. There is always a finite list of countermeasures that are effective against known classes of attacks, but someone always invents new classes.

Ubiquity: Boy, that is pretty discouraging. It seems like a lot of people think that security is hard to achieve, that only certain experts know how to achieve it, and that only certain system platforms can provide it. That leaves the individual feeling stranded and powerless, does it not?

SL: While perfect security is hard, if not impossible to achieve, a high level of practical security is quite attainable. Think of banks. Banks are high value targets for criminals. While robberies do occur, the risk is minimized through enhanced security measures. When banks apply prudent and well-understood measures, they minimize their loss and provide safety for their customers. In the case of software security, we have many well-understood techniques that enable developers to build attack-resistant systems and enable users or system administrators to configure and operate their systems more securely. Things like built-in firewalls and automatic installation of security updates are widespread, easy to use (often available by default) and highly effective. The key is not to seek practical rather than perfect security.

Ubiquity: You have been advocating programming practices that make individual programs more secure. Please say more.

SL: The Security Development Lifecycle (SDL) aims to reduce the number and severity of vulnerabilities in software. We define the SDL in terms of seven stages: training, requirements, design, implementation, verification, release, and response. With each stage we have one or more security practices, for a total of 17 practices in all.

Most of our practices are documented for customers on our SDL website. We recommend threat modeling and attack surface analysis of software designs, the use of safe compiler options, static analysis, and attack mitigation measures, and extensive fuzz testing of parsers and interfaces. There are details, tools, and documentation on the web site.

Buffer overruns are one of the threats. There is a really simple fix for buffer overruns in the unmanaged code C or C++ languages. The simplest buffer overruns come from using the original string handling functions in the C language. The new versions of the string handling functions prevent those simple buffer overruns. We tell people to go back to their old code and replace the old string handling functions with the new ones. If they are developing new code, we tell them to use the new string functions.

Ubiquity: What is fuzz testing?

SL: To help detect whether a buffer overrun potential exists in the code, we recommend the practice of "fuzz testing." We tell developers to feed their software inputs that are generated as randomized variations on valid inputs. Fuzz testing has proven especially effective at finding buffer overruns because it can explore a lot of the possible coding errors that a human code reviewer wouldn't think of and a code analysis tool might not discover. Examples of randomized variations might be to take a length field in a message and make it exceed the allowed size, or to make a length specification inconsistent with the incoming data. There are tools for generating the randomized variations—we provide a simple "file fuzzer" on the SDL website, and there are lots of commercial and free tools available as well.

Another example is SDL practice 7, threat modeling, in the design stage. The idea is to enumerate the classes of attacks likely to be targeted at a system. Developers can then design and implement means to mitigate those attacks. This process enables us to detect design as well as implementation problems—either the omission of necessary security features or failure to include necessary checks. For example, threat modeling might call out a case where an unencrypted communication channel could lead to data disclosure or modification. Or an attacker can delete or alter a file with no access controls.

We have a lot of material on our web site. It goes into detail about all the practices and it provides sample materials to help understand and learn them.

Ubiquity: You give all that away for free?

SL: Yes! We want customers' use of the Internet and Microsoft products to be safe and secure. A lot of people develop software that runs on Microsoft platforms, and customers don't distinguish between a third-party security problem and a Microsoft security problem. We want all our developers to develop securely. Making our tools and guidance free is a good way to help those developers create safer and more secure computing experiences for everyone.

Ubiquity: What kind of results have developers achieved with these methods?

SL: We compare numbers of externally found vulnerabilities in different product versions as a way of telling what kind of progress we and other organizations are making with the SDL. Dan

Kaminsky, a well-known independent security researcher, used fuzzing to compare different versions of Microsoft Office. In his experiment, he found that the number of exploitable or probably exploitable crashes dropped from 126 in Office 2003 (pre-SDL) to 12 in Office 2007 to seven in Office 2010.

The SQL server database product saw vulnerabilities go from 34 in SQL Server 2000 to three in SQL 2005—a 91 percent decrease. The resulting number of vulnerabilities is very low compared to other database products in the market. The more recent SQL Server 2008 has continued the experience—only three vulnerabilities since its release.

We also track the "exploitability" of vulnerabilities that remain in software developed using the SDL. Our "exploitability index" measures the difficulty of building a reliable attack that exploits a vulnerability. We recently found applying the SDL has reduced the exploitability of vulnerabilities in newer versions of Microsoft products by about 30 percent compared to older versions where those vulnerabilities were present.

We are continually updating the SDL process that we use, at Microsoft with the aim of making newer product versions more secure. It's great to see how well the SDL methods work in practice. They are powerful!

Ubiquity: Do you think customers feel safer with the newer systems developed with SDL?

SL: I don't know whether more people would *feel* safer if there were more use of practices like the SDL, but they'd actually *be* safer. Dan Kaminsky's longitudinal study, which compared different product versions, shows the benefits of applying the SDL and so do the product vulnerability counts that we maintain. Attackers know this, and we now see them going after applications that have been built without the use of practices like the SDL.

Ubiquity: Given that security is now recognized as a constant struggle between attackers and defenders, it would seem that SDL techniques are inherently limited because programmers would have to go back and reprogram around new attacks. What about the class of security methods that are based on analogies with the immune system in one's body? Can they respond to new attacks not known at the time the programs were written? Do you think these have potential?

SL: In our experience, application of the SDL helps to blunt new, unanticipated attacks. For example, the SDL requires developers to make stack and heap memory non-executable. Then, no matter what technique an attacker uses to get code on the stack, it won't run.

As to immune system models, I've been reading about analogies to the immune system for more than 20 years, but I've not yet seen software that can recognize new attacks and adapt "on the fly" to block them. This has been a research area for a long time. I don't see that it's come out of the lab yet.

Ubiquity: Thank you.

SL: You're welcome.

About the Author

Peter J. Denning is the editor-in-chief of *Ubiquity* and a past president of ACM (1980-82). Currently he is a distinguished professor, chair of the Computer Science Department, and director of the Cebrowski Institute at the Naval Postgraduate School in Monterey, CA.

DOI: 10.1145/2213616.2213617

