Security

Is Harder than You Think

JOHN VIEGA AND MATT MESSIER, SECURE SOFTWARE

It's not just about the buffer overflow.

Many developers see buffer overflows as the biggest security threat to software and believe that there is a simple two-step process to secure software: switch from C or C++ to Java, then start using SSL (Secure Sockets Layer) to protect data communications. It turns out that this naïve tactic isn't sufficient. In this article, we explore why software security is harder than people expect, focusing on the example of SSL.

HOW WE GOT HERE

Although languages such as Java give programmers fewer chances to shoot themselves in the foot than C does, there is still plenty of opportunity to take off some toes. In an informal study based on security reviews of commercial code, we have seen that C code tends to have five to 10 times more vulnerabilities than Java code. Considering how many vulnerabilities we tend to find in C code, that's still not saying much for Java.

Security problems in software are such a big issue that Microsoft spent two months in early 2002 with all of its developers focused solely on the topic (a major component of the "Trustworthy Computing" initiative). No new features were allowed, just improved security. Developers got extensive training, and executives promised to delay products rather than release them with security problems.

Security

Is Harder than You Think

This wasn't just a one-time push, either. Microsoft has acknowledged that its ongoing push for better software security has led to slipping deadlines.

Despite all the money it has poured into the problem, the big payoff is still out in the distance somewhere. Although the flow of security flaws found in Microsoft products has slowed down since the push, it hasn't come close to stopping yet and—won't for some time.

Whatever Microsoft's investment, it's pretty expensive. Although the company would certainly like to turn public perception about its security practices on its head by going from worst to first, it is unfathomable that there won't be security risks in Microsoft products 20 years from now, despite its mammoth investment.

It's just not possible. Security is all about managing risks, and some risks never go away completely. For example, all encryption schemes have theoretical limits, and there's always some risk that a government or someone else will break it. This example is on the esoteric side, though. In practice, some high-profile risks such as buffer overflows have pretty straightforward solutions, but the easiest classes of attacks to implement—particularly social engineering and insider attacks—do not.

These risks are alarming, but even security-conscious development teams ignore the problem. According to Gartner, 70 percent of attacks on IT systems come from insiders. Yet, most development organizations don't recognize this, perhaps because most IT organizations take a "can't happen here" attitude. Most people adopt the mantra, "We trust our people," but even if that's true, it's not just your people who may be insiders: contractors, friends, and janitors are a risk.

THE PEOPLE PROBLEM

Social engineering is just as difficult to thwart. When a help desk spends all day dealing with users who legitimately forgot their passwords and need them reset, how difficult must it be to weed out people who sound convincing? Even asking for a mother's maiden name or the last four digits of a social security number is not very effective. Good social engineers can often pose success-

fully as authority figures, maintenance workers, or whatever is necessary to achieve their goals. Kevin Mitnick's famous hacking exploits were largely feats of social engineering. If you don't think of social engineering as a serious risk, then we recommend that you read his book, *The Art of Deception: Controlling the Human Element of Security* (John Wiley and Sons, 2002).

In the case of password systems, it would be nice to give them up altogether for the sake of security. But we're probably stuck with them for some time to come, as other solutions, such as biometrics and smart cards, are either too costly or too inconvenient for the user.

Even if these problems had straightforward solutions, the industry still has a long way to go. Too many things can go wrong for the average developer to truly understand all possible risks, particularly when most development organizations value new features far more than security. Gaining insight on every significant risk to software is not an easy task, especially considering some of the subtle things that can go wrong in complex areas such as cryptography.

At the end of the day, the developer shouldn't have to know very much about security. Risk analysis should be left to specialists. The average developer should be given abstractions that make it easy to build good applications without having to understand security risks in detail. They need to understand only how to use the abstraction, at least as much as possible. So far, the security industry has not done the best job of this.

SECURITY AS A NO-BRAINER

SSL is a good example of how the security world could be doing much better. Most developers believe that SSL is a drop-in replacement for standard network sockets—that is, just replace standard API (application programming interface) calls with SSL-enabled calls, and by magic, there's security. Some developers may realize that the system administrator on the server side of a client-server application has to do something to make this all work, but that still tends to be transparent to the developer.

Maybe it should be that easy, but it isn't. SSL is intended to provide an authenticated, encrypted communications channel, where the attacker cannot tamper with data in transit without being detected on the receiving end. Integrity and encryption are the easy part. The difficult part comes in setting up an SSL connection, particularly in performing the authentication. In most client-server cases, the client wants to know for certain that it is talking to the correct server, and the server wants to know for certain that it knows which user is on the other

end of the connection—that is, the parties really want *mutual authentication*. Most people expect SSL to authenticate the server to the client, and then, once an encrypted data channel is established, implement a (usually weak) authentication mechanism over it, so that the server can establish the client's identity.

Unfortunately, the SSL libraries that people use every day don't do adequate server validation by default. In most cases, SSL uses a certificate-based validation scheme for server authentication (other options, such as password-only protocols, are possible, but only experimental libraries implement such things). In such a scheme, the party to be validated (usually the server) presents the other party (the client) with its *certificate*, which is a bundle of information that includes a public key, identifying information, dates denoting validity periods, and one or more digital signatures that serve as endorsements as to the validity of the certificate. The digital signatures are generally put there by a CA (certification authority) such as VeriSign, which is responsible for making sure that it endorses only those certificates that really do belong to the intended owners.

CONSIDERING THE CLIENT

The idea here is that the client can check the signature. If the signature is valid and was put there by a trusted authority, then the client has some assurance that the contents of the certificate are reasonably accurate. Of course, certification authorities are businesses that are rewarded for being efficient, so it should be no surprise that they sometimes issue false certificates. For example, a few years ago VeriSign signed two certificates purporting to come from Microsoft that most certainly did not.

Let's just assume for a minute, however, that the certification authority never makes a mistake. Let's also assume that the companies we care to do business with are good at protecting the private key associated with that certificate, since stealing the private key results in an attacker's ability to impersonate the server to which that certificate is bound.

The client still needs to do several things to make sure that the certificate is the right one. If you want to talk to amazon.com, checking for a signature from VeriSign isn't enough. What if you get a certificate VeriSign signed for Barnes and Noble? Or attacker.org?

Security

Is Harder than You Think

In most cases, the client should be doing the following at the bare minimum:

- Checking to see that the certificate is signed by a known CA.
- Checking to see that the certificate is current (particularly, that it hasn't expired).
- Checking to see that the certificate is bound to the entity that the client wants to communicate with.

The first and third items are of critical importance. If a trusted party didn't sign the certificate, then anybody could have signed it. If a certificate purports to belong to Microsoft, but is signed by `attacker.org`, should you trust it? Probably not. To do this properly, you need a set of trusted credentials from certification authorities, called root certificates. These days, commercial operating systems such as Windows and OS X come with a set of root certs, as do most third-party SSL libraries (including the popular OpenSSL).

VALIDATION

Validating the data within a certificate is not all that straightforward. Usually, the certificate will be bound to a domain name. In some cases, the domain name will point to a single machine. In other cases, the certificate is intended to be valid from any machine in that domain.

For whatever reason, none of the major SSL libraries performs any of this validation for the developer by default. In fact, implementing all three of the previously mentioned simple policies tends to be exceptionally complex.

The situation gets even worse if you want to take into account the fact that valid certificates may get stolen. You either need to download and check the CRLs (certificate revocation lists) that each certification authority issues or check an ordained OCSP (online certificate status protocol) server. Standard crypto libraries make no effort to make it easy to do either of these tasks. They don't keep information on any of these resources. Making matters worse, most CAs don't publicize these resources, anyway. If you do happen to find them, implementation tends to be complex. Few libraries have any

support for OCSP at all.

This is all particularly disheartening, in that most books on secure programming fail to explain these issues. They simply say, "Use SSL," and that's the bulk of their discussion on cryptography. People using OpenSSL can check our book, *Secure Programming Cookbook for C and C++* (O'Reilly, 2003) for code implementing these policies, but so far, people using other platforms are out of luck.

This problem is completely unnecessary. When a user asks to open a client socket, the SSL library could easily perform every reasonable check on the server certificate, including checking to see whether the certificate is bound to the domain supplied by the user. If that kind of validation is somehow too restrictive for some scenario, then there should be a way to circumvent it. Yet, there's absolutely no reason why this couldn't be the default behavior.

STUCK IN THE MIDDLE

The result is that most applications using SSL are subject to man-in-the-middle attacks. This is when the client and server think they're talking directly to each other, when they are actually talking to a malicious proxy. If the user makes no checks, the attacker can present any certificate at all to get the user to talk to it instead of the server. If the user is checking for CA endorsement, but not the entity information, then the attacker can use any certificate endorsed by the CA in question and the client will simply accept it. And, if the user checks the entity information but is not looking for the CA endorsement, then the attacker can construct his own certificate with the correct entity information and present it to the client.

This problem is widespread. It's not just development libraries that get this wrong. Plenty of other people in the industry do, as well. We will illustrate with a real-world company (which we will call "Company X" to preserve its anonymity) that manages financial transactions and provides vendors with a secure solution using SSL. The basic idea is that a vendor does a transaction with a customer, often in an insecure environment, and then, before finalizing the transaction, connects to Company X's server over SSL and checks to see whether the vendor and Company X agree on the transaction. The vendor supplies all the information; then Company X responds, indicating whether the transaction is valid.

The problem is that Company X doesn't show merchants how to make a secure SSL connection to its server—rather, by way of some awful documentation and sample code, it shows vendors how to make an

insecure SSL connection. In fact, more than two years after it learned of the problem, Company X still distributes sample code that does absolutely no certificate validation whatsoever. Vendors following this recipe may risk man-in-the-middle attacks.

Some people would argue that man-in-the-middle attacks are only a theoretical problem. Company X's argument is that, since the backbone of the Internet gives little opportunity for attacker footholds, these exploits are "highly improbable." Assuming that no one can exploit the Internet's router infrastructure (even though Cisco's IOS software is written in a language prone to buffer overflows, and exploits in IOS have been found before), people can launch man-in-the-middle attacks from machines on the same underlying medium as either of the two endpoints. That is, any machine on the same local subnet as one of the endpoints can be leveraged to launch this kind of attack.

Such attacks are simple to get working. There are tools that automate the interception process, such as *dsniff* (<http://naughty.monkey.org/~dugsong/dsniff/>). All an attacker needs to defraud a merchant is often a foothold on the merchant's network. Sometimes, attacking the merchant's ISP will also work.

Many organizations think that their operational security processes address this problem. Particularly, some people think that switches thwart this interception problem. Unfortunately, a technique called ARP (address resolution protocol) spoofing (where low-level network addresses are faked by an attacker) makes interception possible in a switched environment. In general, you should always assume that the attacker has complete control over the network.

Yes, man-in-the-middle attacks are real, and people launch them. We've seen evidence that people have targeted at least one of Company X's vendors, but, since most of Company X's transactions are done in the clear, merchants aren't likely to be subjected to this particular attack. The bad guys tend to go after the weakest link first.

WHAT CAN WE DO?

The fact remains that we, as an industry, are clearly not doing a good enough job understanding and mitigating the risks in our software. Vendors have largely failed to provide the right abstractions for the developer, and the right mental model to make sure those abstractions are used effectively. Developers ultimately shouldn't have to know very much, if anything, about certificate validation, SQL injection attacks, buffer overflows, shatter attacks,

and so on. They should be hit over the head with good abstractions, and the minimum amount of knowledge necessary to use those abstractions properly and to identify what they need to know about security beyond those abstractions.

Because the real world isn't yet very kind to development organizations, however, they need to be far more diligent about security. Since the world isn't doing a good job educating people about risks, organizations should develop as much expertise in this area as they can. Even when the industry has provided great APIs that keep us from having to work too hard to protect ourselves against man-in-the-middle attacks, buffer overflows, SQL injection attacks, integer overflows, cross-site scripting attacks, session fixation attacks, and so on, we should never stop thinking about what might go wrong.

It's unlikely that there will be absolute solutions to protect us from all the threats facing our software, particularly when considering insider risks and social engineering, but education, awareness, and diligence can help get us there. For example, financial institutions spend significant resources on thoroughly researched risk analyses up front, and doing so yields demonstrable results. Many development organizations, Microsoft included, are beginning to use lightweight threat modeling and risk analysis techniques such as attack trees.

Techniques such as attack trees, recipes for avoiding common mistakes, and other best practices are now well documented in books such as *Building Secure Software* (John Viega and Gary McGraw, Addison Wesley Professional, 2001), *Writing Secure Code* (Michael Howard and David LeBlanc, second edition, Microsoft Press, 2002), and the *Secure Programming Cookbook for C and C++* (John Viega and Matt Messier, O'Reilly, 2003). We recommend taking the time to study such resources, instead of the traditional approach of ignoring the problem! ☹

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

JOHN VIEGA is the chief technology officer of Secure Software (www.securesoftware.com) and the coauthor of three books on software security, including *Building Secure Software* (Addison-Wesley, 2001) and the *Secure Programming Cookbook for C and C++* (O'Reilly, 2003).

MATT MESSIER is director of engineering at Secure Software and the coauthor of the *Secure Programming Cookbook for C and C++* (O'Reilly, 2003) and *Network Security with OpenSSL* (O'Reilly, 2002).

©2004 ACM 1542-7730/04/0700 \$5.00