**Solutions to many of our security problems
already exist, so why are we still so vulnerable?**

JOHN VIEGA, SECURE SOFTWARE

Security
Security
Security

Problem Solved?

There are plenty of security problems that have solutions. Yet, our security problems don't seem to be going away. What's wrong here? Are consumers being offered snake oil and rejecting it? Are they not adopting solutions they should be adopting? Or, is there something else at work, entirely? We'll look at a few places where the world could easily be a better place, but isn't, and build some insight as to why.

WHY CAN'T WE BEAT BUFFER OVERFLOWS?

One problem that's on the radar screen for most techies is the buffer overflow. A buffer overflow occurs when a program writes more data into a memory buffer than was allocated for that buffer. In C, C++, and assembly language, the program will just keep writing over whatever memory happens to be next. If what happens to be next is an address that points to executable content, then attackers can overwrite that with a pointer to their own code. For example, if you have a buffer allocated on the call stack, an attacker might be able to write over the saved instruction pointer an address that tells the system where to go when the current function returns. If the attacker also sticks some code in there, and then overwrites the saved instruction pointer with the address of that code, the attacker's code will run. That code can be crafted to do pretty much anything. It's common for attackers to give themselves a command prompt or install software for remote computer control.

Problem Solved?

Buffer overflows are one problem that the world seems to know how to solve, as evidenced by languages such as Java, C#, and Python that are not susceptible to the issue. On the surface, the reason why we still are plagued by the problem is obvious: we still use C, C++, and assembly in a heck of a lot of applications.

The security industry complains about how development organizations are too eager to give up on high-level languages in the name of efficiency. That's true, to some degree, but there are a lot of legitimate needs for writing code in low-level languages. Sometimes the concern really is efficiency (for example, you wouldn't want Microsoft writing its entire operating system in C#). Sometimes there is functionality that can be accessed sanely only through a low-level API. This is often the case when writing to a large legacy code base, whether extending it or maintaining it. Then, there is something to be said about the cost of transitioning teams of expert C programmers to a language they don't know nearly as well, simply for the sake of security.

At the end of the day, not using C, C++, or assembly is only a partial solution. Security requirements aren't the only requirements a system will have, and usually aren't even the most important ones. Thankfully, most of the security industry has recognized this and looked for solutions for those environments.

The obvious thing to do would be to add automatic bounds checking to the compiler, which would help in cases where efficiency isn't a big deal. But the perception, right or wrong, is that it would be a big deal, so there are a wealth of technologies that try to protect against the problem in more efficient ways:

String-handling libraries. Examples include the C++ String class and the SafeStr library for C programs (www.zork.org/safestr). These require diligence on the part of the programmer, who could forget to use the APIs. There are still cases where these APIs can be misused, if one isn't careful. Also, plenty of buffer overflow problems don't involve string handling, some of which can be exploitable.

Static analysis tools. These can look for not just buffer overflows, but also a host of other security problems in applications. Some good tools are becoming commer-

cially available. Generally, however, you need to have the source code to get quality results, it can take resources to actually mitigate potential problems, and not every development environment is supported with good tools.

Nonexecutable stacks. One of the first technologies introduced, nonexecutable stacks take away the easiest vector for turning a buffer overflow into an exploit, but it is rarely the only way—not to mention that this doesn't help at all if the buffer that can be overflowed is allocated anywhere other than the execution stack (e.g., dynamically allocated).

Canary-based techniques. These techniques, such as StackGuard and the /GS flag in the Microsoft Visual Studio compiler, place special values on the stack that should be impossible for an attacker to guess, then try to detect stack overflows by looking to see if those values change when they shouldn't. This doesn't stop every kind of buffer overflow, nor even every kind of stack overflow. In fact, only this year, there was a paper on how to sometimes use the code for the /GS flag against itself.

Randomization techniques. In randomization techniques, important parts of memory, such as the start of the stack and the start of the heap, get randomized when the program starts up. Thus, an attacker who does have a buffer overflow has to do a lot more guesswork about where in memory important things are (particularly, the memory address of the code to be executed, or any other data to be manipulated in order for execution to occur, such as the stored return address). The problem with these techniques is that the space of possible values isn't all that large. In many cases, the attacker can automate the attack, and just keep trying until it works. Even if the attacker has to crash a program 1,000 times before exploiting it, that's often not a problem.

Nonexecutable memory pages. This is a feature of the newest AMD and Intel processors. It's very much like a nonexecutable stack, but one can mark any page of memory as data-only. The basic idea here is to lump all your executable code into a few pages that contain no data, then mark all your other pages as nonexecutable. When this technique is used, it can be highly effective, but not every program can use it. Nor is the technique universally effective. Even when a program can't be made

to execute anything not in the code segment, it's still quite possible to do malicious things simply by injecting data. For example, one of the common techniques for circumventing these mechanisms is to craft malicious data to important system calls. On a Unix system, the common technique is "return into libc", where the programmer uses the buffer overrun to have execution jump to a standard library call such as `exec()`. Also, they set up the execution stack so that when these things are called, they're called with the attacker's malicious arguments. Basically, this technique allows the attacker to run any program on the machine with the same privileges as the vulnerable program. And, the attacker gets to pick how the program is called.

Some of these techniques need to be applied at compilation time, particularly canary-based techniques. This makes it difficult for open source developers to be sure that their programs are secure, since it is usually the user compiling the code. Most of the other techniques have similar problems in that they rely on the end user to be on a particular hardware platform and to be running a particular operating system version.

REAL-WORLD REQUIREMENTS

It's true, development teams may value efficiency a bit more than they should. For teams that do, there is now a research project providing a safe version of C called Cyclone. If this kind of solution gets industrialized, it might strike a balance that is better from the point of view of the developer. Unfortunately, it will probably be a long while. The safe languages will probably stay significantly less efficient than assembly and the fastest languages.

It's also clear that security is rarely the most important requirement for development organizations. The most secure solution is often not the most desirable. When you can't do what's most secure, however, the drop-off in how much security you can easily obtain can become incredibly steep.

A good example of this is authentication systems. This is another area where we have pretty good solutions. The security community will recommend you use multifactor authentication, which could include biometric scanning, cryptographic tokens held on smart cards, and passwords. If all of these mechanisms are in place and are always applied, then you can end up with a really strong system. There are a few problems, however. You'll end up with an expensive solution (and it could be too expensive if you want to have the entire Internet as your customer base—you're not going to want to pay to put a fingerprint

scanner on everyone's desk). Also, many people will find the resulting system unusable. It can be a huge hassle to have to do all those things—and users will complain, look for workarounds, and even refuse to use the system.

People want security, but they also want their systems to be easy to use. As a result, they will prefer solutions that are less secure than optimal. Even having to manage lots of authentication credentials (such as passwords) can be a chore—thus, the growing market for single-sign-on solutions.

Let's look at passwords. To provide the best possible security, they should generally be hard to guess, and the same password should not be used in multiple places. The problem is, users hate picking passwords that are hard to guess, because then they're more likely to forget them. They're also likely to use the same password in multiple places, no matter what you tell them. Some people write passwords down, which security experts try to discourage—it is easy for someone to look under your keyboard for that slip of paper you left there.

In high-assurance areas, such as the financial community, the misery continues for the end user. Passwords need to be changed every couple of months, all in the name of minimizing the window of vulnerability, and you can't cycle through a few favored passwords, because these systems keep enough of a memory to detect when you're reusing something.

People want security, but they also want their systems to be **easy to use**. As a result, they will prefer solutions that are **less secure than optimal**.

Problem Solved?

Another problem is online password guessing. EBay would like to stop attackers from writing programs that try to guess passwords of users by brute force, by attempting to log in until they get the password right. This is usually a highly effective technique, since there are so many bad passwords out there. To prevent this, some systems allow only a limited number of attempts in a short period of time. As a result, people lock themselves out of their own accounts all the time. This can end up being a denial-of-service attack.

USABILITY AND SECURITY

It's clear that security often trades off with other important requirements, especially usability. Does it always have to be that way? Actually, it doesn't. Usability and security can go hand in hand, even with a password system. Particularly, one class of password protocols, called zero-knowledge protocols (see sidebar), avoids almost all of the major problems of traditional password systems. These protocols are carefully designed not to leak any unnecessary information about the password to attackers.

For example, with traditional password protocols, an attacker who can see the protocol messages can store those messages offline and try to replay the protocol using lots of different password choices, to see if one will produce the set of messages that were captured. This can be really easy with password protocols that simply send the password over the wire.

With a good zero-knowledge protocol, this class of attack (called an offline "dictionary" or "brute-force" attack), doesn't work. It is still possible to guess, particularly by trying to log in to the server directly with a guess. The difference is that the attacker needs to participate in the protocol once for each guess. There's nothing the attacker can do with offline information.

It's easy to limit the number of attempts to use an account. While this kind of throttling can lead to denial of service, it can make even a very weak password really strong, because attackers are extremely limited in the number of guesses they can take. For example, if my password is publicly known to be a three-letter word, all in lowercase, there aren't many possibilities. But, if the attacker gets only one guess, the odds are still minuscule.

With a scheme like this, users need just a little bit of coaching to come up with good passwords.

Nonetheless, zero-knowledge password protocols aren't widely used, and this is partially the security community's fault, as many people in the community don't even know they exist (including security vendors). The people who do know about them tend not to promote them, because there are patents covering most of the space. The most popular protocols such as SRP (Secure Remote Passwords) are covered by several patents. Only a few obscure protocols such as PDM (Password-Derived Moduli) may be free of these restrictions, but there is enough uncertainty that many people simply avoid the whole class of protocol and look for other ways to mitigate their risks. These solutions should be better promoted and better supported with libraries. Software development organizations should determine whether they want to license patents or use something like PDM that may be patent-free, with the option of replacing it if someone makes a strong claim at a later date.

Of course, even if you were to use zero-knowledge password protocols, or even stronger authentication systems, there would still be risks. One of the biggest risks is social engineering: for example, a user convincing someone in tech support to reset a password. Humans are often the weakest link in the chain, so the security community tries to take them out of it, whenever possible. For example, with passwords, security people are starting to push the notion called *personal entropy*, where personal questions—such as, "What's the name of your favorite pet?"—are used as a backup form of authentication. The security community doesn't do enough yet to teach people how to apply this technique well. In particular, you can't have a small set of questions, because it's often not hard to find out someone's mother's maiden name or the names of their pets. A high-profile example of this occurred when Paris Hilton's cellphone was "hacked." An attacker was able to get into Hilton's online T-Mobile account by knowing that her favorite pet is her dog Tinkerbell, which is public knowledge. Still, using arcane personal information can be effective, particularly for users that aren't as high profile as she is.

Social-engineering attacks often don't have to be too

Zero-knowledge Basics

Consider a client-server scenario where Alice wants to talk to her bank using a zero-knowledge password protocol. In such a protocol, Alice and the bank share one thing in common, Alice's password. Actually, the bank won't store her password directly. Instead, it will store some function of her password, as is best practice for password storage. This value is called the *validator*, because the bank will use it to validate that Alice is who she says she is.

Traditional password protocols either send a password directly over the network or send some function of the password. In such cases, if Mallory (the attacker) can capture the network traffic, she will either get the password directly or be able to run a brute-force "crack" attack.

Zero-knowledge protocols do not send the password over a network, nor do they send any data that gives away information about the password. Yet, Alice and the bank can still prove to each other that they both know the password, without Alice having to reveal her password.

This is nonintuitive, but we can explain it using a classic real-world example called Ali Baba's cave (figure 1). The line between R and S is a door that can be opened only with the

secret password. The bank wants to prove that Alice knows the password, but Alice doesn't want to reveal any of the password if she's not really talking to the bank. They can do this with the following protocol:

The bank starts by standing at P, while Alice goes to either R or S. Alice chooses R or S, but does not reveal which one. The bank has no way of knowing, because it cannot see Q. When Alice has had enough time to commit to her choice, the bank moves to Q. It then randomly chooses either left or right, and calls this out. Let's say it calls "left." If Alice is already in the left tunnel, she simply walks out. If she's not in the left tunnel, she whispers the secret password to the door, so that the bank cannot hear her. She then walks through the door and up the left tunnel.

If Alice comes out the right side, the bank is certain that the person does not know the password. If Alice comes out the left side, there's no way of determining whether she knows the password, because she had a 50 percent chance of getting it right.

If they repeat this protocol 20 times, however, the odds of Alice getting each challenge right without knowing the password are worse than one in a million. That is, if Alice comes up the proper path 20 different times, the bank can be quite confident that she knows the password.

If the challenger isn't the bank, but is instead Mallory, Alice does not reveal any information about the password. She does prove that she knows the correct password, of course; otherwise, she would never be able to answer the challenges.

Note, however, that if Alice doesn't know the password, she still gets to try at least one password out, possibly more. In a real scheme, it is possible to limit Alice to one guess for the whole protocol. Similarly, if Alice is legitimate, but is somehow tricked into trying to authenticate to Mallory (for example, with a so-called phishing attack), then Mallory will still be able to take only a single guess. In general, Alice will issue Mallory a "challenge," where a correct answer to the challenge demonstrates knowledge of the password. Mallory gets no information about the password from Alice, except whether her guess was correct.

Problem Solved?

sophisticated to work. For example, if you call someone, claiming to be with a legitimate research organization doing a study on how people use and choose passwords, most people will divulge their PINs and passwords, particularly if there's an offer to get paid for participation. All it takes, in most cases, is an air of professionalism. This is a problem that doesn't seem to have an ideal technical solution. The security industry recommends using multiple types of authentication (called multifactor authentication), where one factor is generally passwords, and the other factors are often smart cards (including the popular RSA SecurID product) and biometric devices. This way, even if the password is compromised, there's another layer of authentication that the attacker would need to find a way to compromise.

The problem with multifactor authentication mechanisms is that every solution beyond passwords has some sort of real-world constraint that keeps people from wanting to use it. Often, this is cost, particularly with biometric devices. Even token-based systems such as SecurID or Java-based smart cards have a significant infrastructure cost for cards and readers that are expensive to deploy, and essentially impractical if you're writing software for the whole Internet to use. It's possible to add another authentication factor without physical devices by using PKI (public key infrastructure) credentials. PKI simply needs to live as a pile of bits. Users have a hard time with these systems, however, because they make it difficult to migrate between machines.

Since passwords will probably always be the most popular option for authentication, it seems reasonable to keep looking for both technical and educational methods to keep people from disclosing their passwords. For example, researchers may want to figure out how to discourage people from disclosing their passwords to system administrators when they're having trouble logging into a system. This might not require any major technical breakthroughs, but simply specification and standardization. For example, we can easily build a protocol where users can set up temporary passwords for their own accounts, which they can then give to system administrators. These passwords can be randomly chosen, short, and simple. If these kinds of zero-knowledge and password-reset pro-

ocols were standardized and widely deployed, we could legitimately tell users that they have no reason to disclose their passwords under any circumstances.

IF WE HAVE SSL, WHY IS IT HARD TO SECURE DATA COMMUNICATIONS?

All in all, there are secure and usable solutions available for many security problems, but the security community isn't good enough at usability to give users the functionality it could and should without major security trade-offs. For example, look at SSL/TLS. The end user wants an abstraction that is as close to being a drop-in replacement for traditional sockets as possible. SSL appears to work that way—the code will be functional, but, unfortunately, it won't be secure. No SSL socket libraries perform the proper authentication checks to build a secure connection. Some might check to see that the server certificate hasn't expired, and they might even check to see if that certificate is signed by a known certification authority, but none of them checks to see if the certificate actually maps to the host that the client tried to connect to. You either have to install your own white list of valid certificates or validate the content of the certificates manually, which generally takes a fair bit of code. There's absolutely no reason, though, why this couldn't be a default in SSL libraries, with hooks to allow you to customize your checking if you have special needs.

SSL introduces other annoyances, however, such as the need to get a signed certificate for the server and to build a client authentication system on top of the SSL connection to make sure both sides are authenticated to each other. Unfortunately, the end users end up thinking their security requirements are satisfied when they're not, because they don't understand the problem.

Why should they have to? If there are better solutions that require less understanding, then the security community should do a better job of providing them and making the end user aware of them. For example, zero-knowledge password protocols actually authenticate both sides and do a cryptographic key exchange. It would be easy to use such a protocol in conjunction with a general-purpose method for protecting data on the wire, providing users with a truly simple interface to a "secure

Security

Problem Solved?

channel.” This would meet the security needs of most applications, particularly if built upon standard components such as the AES (Advanced Encryption Standard) block cipher (standardized by the National Institute of Standards and Technology). Several block cipher modes of operation could be used with AES to provide the underlying security guarantees (e.g., GCM, which is currently standardized in IPsec, and on track for standardization in 802.1ae).

The problem is that there needs to be a single simple interface to bundle everything together, from the authentication protocol to the encryption and authentication of ongoing data. This should be done while also eliminating common risks. For example, modes such as GCM can protect against a class of attack known as capture-replay if they are used correctly, but not everyone has the need to protect against such attacks, so it's not mandated that you avoid them. As a result, it's possible to use GCM without protecting against this kind of attack. The goal here is to have everything bundled into a library that is simple enough that there would be less concern about the latent security risks in the underlying library (there are several crypto libraries that, while good for crypto, aren't well written in terms of avoiding other security concerns, such as buffer overflows).

WHAT ELSE IS WRONG?

There are plenty of other problems that the security industry could address well by providing simple, usable interfaces to dangerous operations. For example, there should be common frameworks for protecting against SQL injection attacks, cross-site scripting attacks, and so on. At the end of the day, usability is a fundamental skill the security industry should have, but has always lacked. Usability isn't the only problem, however.

First, the commercial security sector is, quite understandably, focused too much on the bottom line to be giving customers what is in their long-term best interests. That is, they are focused on low-hanging fruit that can improve the world, instead of coming up with fundamental solutions to the problem. For example, despite e-commerce taking off in 1997, to mitigate software vulnerabilities, the commercial world has, until recently,

focused exclusively on network security mechanisms such as firewalls and intrusion detection systems. Network-based solutions are easy to build, but cannot be a complete solution to problems.

Ultimately, we have to get people writing more secure code. This requires giving them better educations and better abstractions to make the job fundamentally easier, while still meeting the business requirements that development shops are really going to have. The corporate world, however, has started moving in this direction only in the past two years. To be fair, even the academic world didn't really start looking at the problem in any depth until this decade, but then again, that community is largely driven by the short-term needs of industry, and also tends to work on a low-hanging fruit strategy.

The academic community that does go after core security solutions isn't completely in the right place, either. It needs to be more concerned with balancing security with real-world requirements and concerns. For example, the first tools to address unknown software security problems such as buffer overflows were Web-based scanners. Lots of organizations bought these tools on the promise of largely automating something that they couldn't previously do (manual reviews are too expensive, and too difficult to do well, because not many people have the breadth of expertise, the patience, or the memory to understand enormous, complex systems and their interactions).

The companies that bought this first wave of tools learned that there were huge hidden costs in using them well. First, the tools tend to produce a ton of output, but figuring out what to do about a problem requires a human being. Prioritizing this output is a difficult task, particularly since it's common for most of the problems to be false positives. Further, since the tools are black-box tools, they don't give great insight into where in code a problem might lie, meaning that a significant amount of legwork is often required.

Ultimately, businesses are wrestling with even more fundamental issues that the security community has been slow to answer. For example, development shops can now find a set of tools to help them address software security, but they still generally don't understand how they should

Security

Problem Solved?

address the problem as a business. Where, when, and how should these tools be used (for example, while developing, during daily builds, or during beta)? What other kinds of activities should one be performing that can't be automated with tools, such as security reviews when originally designing software? What should the relative investment be in these various activities?

Business process for secure development is extremely important, but isn't as far along as it should be. The first books in this space haven't really dealt with process per se, but are instead collections of best practices.

Thankfully, work in this area is now ramping up. Microsoft and a few others have put some process in place for doing architectural security reviews. As of this year, there is a complete life-cycle process for software security, from requirements extraction to architectural and implementation review, all the way down to testing and deployment: Secure Software's CLASP (Comprehensive, Lightweight Application Security Process). So little work has been done in this area, however, that there are still many open issues that the security community needs to address, such as development of more effective metrics to help in making business decisions about software security.

SECURITY SOLUTIONS DON'T EXIST IN A VACUUM

The security community knows how to solve some really big problems that aren't solved in practice. For example, we've got technical solutions to the buffer overflow problem and know how to solve a lot of practical problems with password usage. We can solve hard problems, but there's clearly some difficulty in deploying solutions that are actually effective.

The fundamental problem is that the security community frequently ignores real-world requirements for systems. Its designs often assume that security is either the only requirement or the most important requirement. There are areas without any explicit requirements for security functionality, but with implicit ones (particularly, usability). In these cases, the security community seems to ignore the impact of these concerns.

This is unfortunate, because usability is a critical consideration in security. If security features aren't easy to use

securely, then people won't use them securely. Similarly, if security functionality is required, but the system isn't usable, then people will look for a way to turn off the functionality, or will use something else. This is a tough balance to strike, and the security industry doesn't, in general, spend much energy trying to find it. Usability is a discipline that seems simple, because designers often feel that their intuitive sense of how things should work is the most usable way. That's rarely the case. An engineering discipline all its own has developed to help designers figure out how to make things genuinely easy to use (see, for example, *Usability Engineering*, by Jakob Nielsen, Elsevier Science and Technology Books, 1995).

The security community needs to widen the scope of the business requirements it is addressing. It needs to understand that technical solutions to security problems don't exist in a vacuum and that it can be hard work simply bundling those solutions in such a way as to minimize the end user's risk over the long term. There's certainly some realization of these issues, but progress isn't being made as quickly as it should be. With very little fuss, we should be able to do simple things such as create a secure network connection or use a database without risk of SQL injection. Until we can do these things, it's up to the rest of the world to try to shape the security world by being demanding customers. Q

LOVE IT, HATE IT? LET US KNOW

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

JOHN VIEGA is CTO and founder of Secure Software, where he is responsible for the company's core processes and algorithms for security analysis. He also works to promote better security practices for developers and is a frequent lecturer on the topic. He has co-authored three books in the field, including *Building Secure Software* (Addison Wesley, 2001), *Network Security with OpenSSL* (O'Reilly, 2002), and *Secure Programming Cookbook for C and C++* (O'Reilly, 2003). Viega is a co-author of GCM, an encryption mode that is in the draft 802.1ae standard. He is the original author of Mailman, the GNU mailing list manager. He holds a B.A. and an M.S. in computer science from the University of Virginia.

© 2005 ACM 1542-7730/05/0600 \$5.00