Software security principles: Part 1

PDF e-mail it!

The chain is only as strong as its weakest link

Gary McGraw (gem@cigital.com), Vice president of corporate technology, Cigital
John Viega (viega@cigital.com), Senior research associate and consultant, Cigital

01 Oct 2000

Contents:

Subscriptions:

In this series of articles, Gary and John offer the 10 most important points to keep in mind when designing and building a secure system. This installment explores the importance of reinforcing the weakest -- and most susceptible -- parts of your system. And it's worth noting that the problem isn't necessarily a weakness in the software.

Software security is certainly a vast, complex topic. One of the biggest challenges to the field is that new types of security flaws that completely defy all known patterns are always possible. For a good example, we need look no further than cryptography, where it is relatively easy to build algorithms that resist the best-known attacks on other algorithms. Protecting against unknown attacks is a far greater challenge. Moreover, there is plenty of room for new attacks to emerge; every few years a significant one does. Another serious challenge is that just keeping up with known problems is difficult, because there are so many of them.

Sure, it's not practical to protect against every possible type of attack. However, we can avoid being overwhelmed with the large volume of knowledge by using a good set of principles when designing and building software.

In the next several installments we will present 10 principles for developing secure software. The goal of these principles is to isolate the most important points you should keep in mind when designing and building a secure system. Following these principles should help you avoid lots of common security problems. Of course, this set of principles will not be able to cover every possible flaw that could show up. We don't pretend that any set of principles could; we are just trying to follow the "90/10" rule -- avoid 90% of the potential problem by following 10 simple rules. Today we'll start out with the first, which is to focus energy on the weakest parts of your system.

Principle 1: Securing the weakest link

One of the most common analogies in the security community is that security is a chain; a system is only as secure as the weakest link. One consequence is that the weakest parts

of your system are the parts most susceptible to attack.

It's probably no surprise to you that attackers tend to go after low-hanging fruit. If they target your system for whatever reason, they're going to take the path of least resistance. That means they'll try to attack the parts of the system that look weakest, and not the parts that look strong. Even if they spend an equal effort on all parts of your system, they're far more likely to find problems in the parts of your system most in need of improvement.

This intuition is widely applicable. There's generally more money in a bank than a convenience store, but which one is more likely to be held up? The convenience store, of course. Why? Because banks tend to have much stronger security precautions; convenience stores are much easier targets.

Let's say that you own an average bank and an average convenience store. Would it be more cost-effective for you to add extra doors to your vault, and double your security staff, or to take the same money and hire security officers for your convenience store? The bank probably already keeps tellers behind bulletproof glass, and has cameras, security guards, a locked vault, and doors with electronic passwords. In contrast, the convenience store probably has a less sophisticated camera system, and little else. If you're going to invest in the security of any one part of your financial empire, the convenience store would be the best choice, because it's at much greater risk.

This principle has obvious applications in the software world, but most people don't pay any attention. In particular, cryptography is seldom the weakest part of a system. Even if you use SSL-1 with 512-bit RSA keys and 40-bit RC4 keys, which are considered incredibly weak cryptography, an attacker can probably find much easier ways in. Sure, it's breakable, but doing so still requires a large computational effort.

If the attacker wants access to the data that travels over the network, then they'll probably target one of the endpoints, try to find a flaw like a buffer overflow, and then look at the data either before it gets encrypted or after it gets decrypted. All the cryptography in the world can't help you if there's an exploitable buffer overflow -- and buffer overflows abound in C code.

For this reason, while cryptographic key lengths can certainly have an impact on the security of your system, they aren't all that important in most systems, where far greater things are wrong. Similarly, attackers generally don't attack a firewall itself, unless there's a well-known vulnerability in the firewall. Instead, they'll try to break the applications that are visible through the firewall, because they're generally much easier targets.

Identifying what you perceive to be the weakest components of a system should be easy if you perform a good risk analysis. You should address what seems to be the most serious risk first, instead of the risk that seems easiest to mitigate. Once it's clear that some other component is a bigger risk, you should focus your efforts elsewhere.

Of course, you could apply this strategy forever, because security is never a guarantee. You need some stopping point. You should stop when all your components appear to be within the threshold of acceptable risk, by whatever metrics you define in your software engineering process.

Social engineering: A common weak link

Sometimes it's not the software that is the weakest link in your system; sometimes it's the surrounding infrastructure. For example, consider *social engineering*, which is when an attacker uses social manipulation to break into a system. Typically, a service center will get a call from a sincere-sounding user, who will talk the service professional out of a password that shouldn't be given away, or some other information that should remain secret. This sort of attack is generally pretty easy to launch, because customer service representatives don't like to deal with stress. If they are dealing with someone who seems to be really mad about not being able to get into their account, generally, they will not want to aggravate the situation by asking questions to authenticate the remote user.

Even if they do ask questions to authenticate the person on the other end of the phone, what are they going to ask? Birth date? Social Security number? Mother's maiden name? All of that information is easy to obtain if you know your target. This problem is a common one, and is incredibly difficult to solve.

One good strategy is to limit the capabilities of technical support as much as possible. For example, you might choose to make it impossible to change a password; if you forget it, then you must create another account. Of course, that particular example is generally not a good solution, because it is a huge inconvenience for users.

The following elaborate scheme is a better example. Before deploying the system, a large list of questions is composed (say, no fewer than 400 questions). These questions should be generic enough that any one person should be able to answer the question. However, the correct answer to any single question should be pretty difficult for anyone other than the respondent to guess. When the user creates an account, we select 20 questions from the list, and ask the user to answer six of them for which he has answers, and is most likely to give the same answer if asked again in two years. Here are some potential sample questions:

- In how many political elections had you voted, as of January 2000 (presidential or otherwise)?
- Name the celebrity you think you most resemble.
- What was your single biggest embarrassment of your high school years?
- What is your single biggest regret of your relationship with your first significant boyfriend or girlfriend?
- Whose death was the first death that was significant to you, be it a person or animal?

When someone forgets their password, and calls technical support, technical support can only refer them to a Web page. The user is given three questions from the list of six, and

must answer two correctly. If the user answers two correctly, then we do the following:

- Give them a list of 10 questions, and ask them to answer three more.
- Let them set a new password.

We should probably only allow any one person to authenticate in this way a small handful of times (say, three).

The result of this scheme is that users can still accomplish what they need to when they forget their passwords, but tech support is protected from social engineering attacks.

Software security principles: Part 2

Defense in depth and secure failure

Gary McGraw (gem@cigital.com), Vice president of corporate technology, Cigital
John Viega (viega@cigital.com), Senior research associate and consultant, Cigital

01 Nov 2000

In this series of articles, Gary and John offer the 10 most important points to keep in mind when designing and building a secure system. Part 1 explored the importance of reinforcing the weakest parts of a system. This time, they look at the next two principles: *Defense in depth* -- which advocates the use of multiple defense strategies -- and *secure failure* -- the notion that system failure is not necessarily the same as system vulnerability.

In this series, we present a set of principles that can help you squash the majority of security problems in your code. Last time, we talked about how to identify and secure the weakest links in a system. This time, we examine how it is important to provide redundant security measures, and why you should be careful not to rely on insecure behavior when parts of a system fail.

Principle 2: Defense in depth

The idea behind defense in depth is to manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense will, ideally, prevent a full breach. This principle is well-known, even beyond the security community; for example, it is a famous principle for programming language design:

Defense in Depth: Have a series of defenses so that if an error isn't caught by one, it will

probably be caught by another. (From Bruce MacLennan's *Principles of Programming Languages*. See Resources.)

Let's return to our example of providing security for a bank. Why is the typical bank more secure than the typical convenience store? Because there are many redundant security measures protecting the bank -- the more measures it has, the more secure it is. Security cameras alone are usually enough of a deterrent. But if attackers don't care about the cameras, then a security guard is there to physically defend the bank. Two security guards will provide even more protection. But if both our guards get shot by masked bandits, then at least there's still a wall of bulletproof glass and electronically locked doors to protect the tellers. If the robbers do happen to kick in the doors, or guess the PIN, at least the robbers will only have easy access to the registers, because we have a vault protecting the rest. Ideally, the vault is protected by several locks, and cannot be opened without the presence of two individuals who are rarely at the bank at the same time. And as for the registers, they can be equipped with dye-emitting mechanisms that mark the bills.

Of course, having all these security measures in place does not ensure that our bank will never be robbed successfully. Bank robberies do happen, even at banks with this much security. Nonetheless, it's pretty obvious that the sum total of all these defenses results in a far more effective security system than any one defense alone.

This may seem somewhat contradictory to the previous principle, because we are essentially saying here that multiple defenses can be stronger than the strongest link. However, there is no contradiction; the principle of "securing the weakest link" applies when components have security functionality that does not overlap. But when it comes to redundant security measures, it is indeed possible that the total protection offered is far greater than the protection offered by any single component.

A good real-world example where defense in depth can be useful, but is rarely applied, is in the protection of data that travels between various server components in enterprise systems. Most companies will throw up a corporate-wide firewall to keep intruders out. These companies then assume that the firewall is sufficient, and let their application server talk to their database in the clear. If that data is important, what happens if an attacker manages to penetrate the firewall? If the data is also encrypted, then the attacker won't be able to get at the data without breaking the encryption, or (more likely) breaking into one of the servers that stores the data in an unencrypted form. If we throw up another firewall just around our application, then we can protect ourselves from people who penetrate the corporate firewall. Then they'd have to find a flaw in some service that our application's network explicitly exports; we can keep a tight reign on that information.

Principle 3: Secure failure
Any sufficiently complex system will have failure modes. These are pretty unavoidable. What are avoidable are security problems related to failures. The problem is that when many systems fail in any way, they revert to insecure behavior. In such systems,

attackers only need to cause the right kind of failure, or wait for the right kind of failure to happen.

The best real-world example we've ever heard is one that bridges the real world and the electronic world -- credit card authentication. Big credit card companies such as Visa and MasterCard spend lots of money on authentication technologies to prevent credit card fraud. Most notably, whenever you go into a store and make a purchase, the vendor swipes your card through a device that contacts the credit card company. The credit card company checks to see if the card is known to be stolen. More amazingly, the credit card company analyzes the requested purchase in the context of your recent purchases, and compares the patterns to the overall trends of your spending habits. If their engine senses anything fairly suspicious, the transaction is denied.

This scheme is remarkably impressive from a security point of view -- until you note what happens when something goes wrong. What happens if the credit card stripe somehow gets demagnetized? Does the vendor have to say, "I'm sorry, your card is not valid because the stripe is broken"? No. The credit card company still supplies the vendor with manual machines that create an imprint of your card, which the vendor can send to the credit card company for reimbursement. If you have a stolen card, it probably won't be authenticated at all. Merchants probably won't even ask you for your ID.

There used to be some security in the manual system, but it's gone now. Before computer networks, you'd probably be asked for your ID to make sure the card matched your license. As another precaution, if your number was on a periodically updated list of bad cards in the area, the card would be confiscated. Also, the vendor would probably also check your signature. These techniques aren't really necessary anymore -- as long as the electronic system is working! If it somehow breaks down, then, at a bare minimum, those techniques ought to come back into play. In practice, however, they don't. Credit card companies feel that failure is way too uncommon in credit card systems to ask vendors to remember a complex procedure when it happens.

When the system fails, the behavior of the system is less secure than typical behavior. Unfortunately, it's really easy to cause the system to fail. For example, it's easy to ruin the stripe of a stolen credit card by running it across a large magnet. In doing so, a thief has more or less minted an arbitrary amount of money, as long as they use the card for small purchases (better validation is often required for large purchases). The good thing about this scheme from a thief's perspective is that failure rarely results in the thief getting caught. Someone can use the same card for a long time this way, with little risk.

Why do credit card companies use such a brain-dead fallback scheme? The answer is that these companies are good at risk management. They can absorb a fairly large amount of fraud, as long as they keep making money hand over fist. They also know that the cost of deterring this kind of fraud would not be justified, because the amount of fraud actually committed is relatively low. (A lot of factors affect this decision, including costs and public relations issues.)

Plenty of other examples pop up in the digital world. Often, the problem arises because of a need to support legacy versions of software that isn't secure. For example, let's say that the original version of your software was naïve, and did not use encryption at all. Now you want to fix the problem, but you have established a large user base. In addition, you have deployed many servers that probably won't be upgraded for a long time. The newer, smarter clients and servers need to interoperate with older clients that aren't updated with the new protocols. You'd like to force old users to upgrade, but you haven't planned for that. Legacy users aren't expected to be such a big part of the user base that it will really matter anyway. What do you do? Have clients and servers examine the first message they receive from one another, and figure out what's going on from there. If we are talking to an old piece of software, then we don't perform encryption.

Unfortunately, a wily hacker can force two new clients to each think the other is an old client by tampering with data as it traverses the network. And worse, there's no way to get rid of the problem while still supporting full (two-way) backward compatibility.

A good solution to this problem is to design in a forced upgrade path from the beginning; the client detects that the server is no longer supporting it. If the client can securely retrieve patches, it does so. Otherwise, it tells the user that they must obtain a new copy manually. Unfortunately, it's important to have this sort of solution in place from the very beginning, unless you don't mind alienating your early adopters.

Most implementations of Remote Method invocation (RMI) have a similar problem. When a client and server want to communicate over RMI, but the server wants to use SSL or some other encryption protocol, the client may not support the protocol the server wants to use. When that's the case, the client generally downloads the proper socket implementation from the server at runtime. This constitutes a big security hole, because the server has not been authenticated at the time that the encryption interface was downloaded. An attacker could pretend to be the server, installing his own socket implementation on each client, even when the client already has proper SSL classes installed. The problem is that if the client fails to establish a secure connection with the default libraries (a failure), it will establish a connection using whatever protocol an untrusted entity gives it, thereby extending trust.

Next time

In our next installment, we'll discuss two more of our 10 design principles. The first is a classic -- the *principle of least privilege*, whose basic premise is to limit access to the minimum required to do the job, and to extend that access for as short a time as possible. The second is almost as well-known, the *compartmentalization priniciple*, which states that you should seek to prevent damage from spreading by building walls between parts of your system.

Software security principles: Part 3

Controlling access: Least privilege and compartmentalization

Gary McGraw (gem@cigital.com), Vice president of corporate technology, Cigital
John Viega (viega@cigital.com), Senior research associate and consultant, Cigital

01 Nov 2000

In this column, Gary and John present a set of 10 principles that can help you remove the majority of security problems from your code. Last time, they talked about providing redundant security measures when appropriate, and how to recover securely from error. This time, they focus on the importance of being stingy when doling out permissions, giving out only what is necessary for as short a time as possible. It is also critical to ensure that a failure in one part of a system does not affect the entire system.

Contents:

Subscriptions:

Principle 4: Least privilege

The principle of least privilege states that only the minimum access necessary to perform an operation should be granted, and that access should be granted only for the minimum amount of time necessary.

When you give out access to parts of a system, there will generally be some risk that the privileges associated with that access will be abused. For example, let's say you go on vacation and give a friend the key to your home in order to feed pets, collect mail, etc. While you may trust that friend, there is always the possibility that there will be a party in your house without your consent, or that something else will happen that you don't like.

Whether or not you trust your friend, there's generally no need to put yourself at risk by giving more access than necessary. For example, if you don't have pets, but only need a friend to occasionally pick up your mail, you should relinquish only the mailbox key. While your friend might find a good way to abuse that privilege, at least you don't have to worry about the possibility of additional abuse. If you give out the house key unnecessarily, all that changes.

Similarly, if you do get a house sitter while you're on vacation, you aren't likely to let that person keep your keys when you're not on vacation. If you do, you're setting yourself up for additional risk. Whenever a key to your house is out of your control, there's a risk of that key getting duplicated. If there's a key outside your control, and you're not home, then there's the risk that the key is being used to enter your house. Any length of time when someone has your key and you're not supervising them constitutes a window of time in which you are vulnerable to an attack. You want to keep such windows of vulnerability as brief as possible, in order to minimize your risks.

Another good real-world example appears in the security clearance system of the U.S. government -- the policy of "need to know." If you have clearance to see any classified document whatsoever, you still won't be able to see *any* secret document that you know exists. If you could, it would be very easy to abuse the secret clearance level. Instead, people are only allowed to access documents that are relevant to those tasks that apply to them.

Some of the most famous violations of the principle of least privilege exist in UNIX systems. For example, on UNIX systems, you generally need to have root privileges to run a service on a port number less than 1024. So, to run a mail server on port 25 -- the traditional SMTP port -- a program needs the privileges of the root user. However, once a program has set up shop on port 25, there is no compelling need for it to ever use root privileges again. A security-conscious program would give up root privileges and let the operating system know that it should never require those privileges again (at least, not until the next run of the program). One large problem with some e-mail servers is that they don't give up their root permissions once they have grabbed the mail port (Sendmail is a classic example). Therefore, if someone finds a way to trick such a mail server into doing something nefarious, it will succeed. For example, if a malicious attacker were to find a suitable stack overflow in Sendmail (see Resources), then that overflow could be used to trick the program into running arbitrary code. Because Sendmail runs with root permissions, any valid attempt by the attacker will succeed.

Another common scenario is that a programmer may wish to access some sort of data object, but only needs to read from the object. Often, however, the programmer actually requests more privileges than necessary, for whatever reason. Generally, the programmer is trying to make life easier. For example, he might be thinking, "Someday I might need to write to this object, and I'd hate to have to go back and change this request."

Insecure defaults might lead to a violation here, too. For example, there are several calls in the Windows API for accessing objects that grant all access if you pass "0" as an argument. In order to get something more restrictive, you'd need to pass a bunch of flags (OR'd together). Many programmers will just stick with the default, as long as it works, because that's easiest.

This problem is starting to become common in security policies for products intended to run in a restricted environment. For example, some vendors offer applications that work as Java applets. Applets constitute mobile code, which Web browsers tend to treat with suspicion. Such code is run in a sandbox, where the behavior of the applet is restricted based on a security policy that a user agrees upon. Vendors rarely practice the principle of least privilege here, because it takes a lot of effort on their part. It's far easier to implement a policy that says, in essence, "let the vendor's code do anything at all." People generally install vendor-supplied security policies, maybe because they trust the vendor, or maybe because it's too much of a hassle to figure out what security policy does the best job of minimizing the privileges that must be granted to the vendor's application.

Principle 5: Compartmentalization

The principle of least privilege works a lot better if your access structure is not "all or nothing." Let's say you go on vacation, and you need a pet sitter. You'd like to limit the sitter's access to just your garage, where you'll leave your pets while you're gone, but if you don't have a garage with a separate lock, then you have no choice but to give the sitter access to the entire house.

The basic idea behind compartmentalization is that we can minimize the amount of damage that can be done to a system if we break the system up into as many isolated units as possible. This same principle is applied when submarines are built with many different chambers, each separately sealed; if a breach in the hull causes one chamber to fill with water, the other chambers will not be affected. The rest of the ship can keep its integrity, and people can survive by making their way to parts of the submarine that are not flooded.

Another common example of the compartmentalization principle is a prison, where the ability of large groups of convicted criminals to get together is minimized. Prisoners don't bunk in barracks, they bunk in cells of one or two. Even when they do congregate -- say, in a mess hall -- other security measures are beefed up to help compensate for the large increase in risk.

In the computer world, it's a lot easier to point out examples of poor compartmentalization than it is to find good ones. The classic example of how not to do it is the standard UNIX privilege model, where security-critical operations work on an "all or nothing" basis. If you have root privileges, you can basically do anything you want. If you don't have root access, there are restrictions. For example, you can't bind to ports under 1024 without root access. Similarly, you can't access a lot of operating system resources directly -- for example, you have to go through a device driver to write to a disk; you can't deal with it directly.

Currently, if an attacker exploits a buffer overflow in your code, that person can make raw writes to disk and mess with any data in the kernel's memory. There are no protection mechanisms preventing that. Therefore, you can't directly support a log file on your local hard disk that can never be erased, which means you can't keep accurate audit information up until the time of a break-in. Attackers will always be able to circumvent any driver you install, no matter how well it mediates access to the underlying device.

On most platforms, you can't protect just one part of the operating system from the others. If one part is compromised, then everything is hosed. A few operating systems, such as Trusted Solaris, do compartmentalize. In such cases, operating system functionality is broken up into a set of roles. Roles map to entities in the system that need to provide particular functionality. One role might be a LogWriter role, which would map to any client that needs to save secure logs. This role would be associated with a set of privileges. For example, a LogWriter would have permission to append to its own log files, but never erase from any log file. Perhaps only a special utility program would be given access to the LogManager role, which would have complete access over

all the logs. Standard programs would not have access to this role. Even if you break a program and end up in the operating system, you'll not be able to mess with the log files unless you happen to break the log management program as well. This sort of "trusted" operating system isn't all that common, in large part because this kind of functionality is difficult to implement. Problems like dealing with memory protection inside the operating system create challenges that have solutions, but not ones that are simple to effect.

More so than a lot of the other principles, compartmentalization must be used in moderation. If you segregate each little bit of functionality, then your system will be completely unmanageable.

Next time
In our next installment, we'll talk about two more principles for writing secure software -- simplicity and privacy. The simplicity principle goes beyond just writing straightforward code; in order to minimize security problems, programs should be easy to use, too. The privacy principle dictates that you should avoid giving out unnecessary information, and suggests that sometimes it's O.K. to lie.

developerWorks.

Software security principles: **Part 4**

Keep it simple, keep it private

Level: Introductory

Gary McGraw (gem@cigital.com), Vice president of corporate technology, Cigital
John Viega (viega@cigital.com), Senior research associate and consultant, Cigital

01 Dec 2000

In this installment, Gary and John present a set of 10 principles that can help you remove the majority of security problems from your code. Last time, they focused on the principles of least privilege and compartmentalization, both of which can help you design high-quality access models. This time around, they'll show you how simplicity is more important when it comes to security than most people suspect. They'll also discuss privacy concerns; giving out too much information can ruin you.

More dW content related to: **software security principles Gary McGraw and John Viega**

Document

Docum
option
requiri
JavaSc
are not
display

Rate this p

Help u this co

Principle 6: Simplicity

In many areas, you're likely to hear the KISS mantra -- "Keep it simple, stupid!" This applies just as well to anywhere else. Complexity increases the risk of problems; this seems unavoidable in any system.

Clearly, your designs and implementations should be as straightforward as possible. Complex designs aren'
easy to understand, and are therefore more likely to have lingering problems that will be missed. Complex
harder to analyze and maintain. It also tends to be far more buggy. We don't think this will come as a big su

Similarly, you should consider reusing components whenever possible, as long as the components you're co
good quality. The more times a particular component has been successfully used, the more you should avoi
This consideration particularly holds true for cryptographic libraries. Why would you want to re-implement
when there are several widely used libraries out there? Those libraries are a lot more likely to be robust than
put together in-house. Experience builds assurance, especially when those experiences are positive. Of cour
always the possibility of problems, even in widely-used components. However, it's reasonable to assume th
involved in the known quantity.

Those bells

It also stands to reason that adding bells and whistles tends to violate the simplicity principle. True enough
bells and whistles are security features? When we discussed defense in depth, we said that we wanted redun
seem to be arguing the opposite. We previously said, "don't put all your eggs in one basket." Now we're say
having multiple baskets." Both notions make sense, even though they're obviously at odds with each other.

What you need to do is strike a balance that is right for your particular project. When adding redundant feat
generally trying to improve the perceived security of the system. Once you've added enough redundancy to
security margin you've decided upon, then you should not add extra redundancy. In practice, a second layer
usually a good idea, but you should carefully consider a third layer.

Despite its obviousness, the simplicity principle has its subtleties. First, you must strive to build as simple a
possible, while still meeting your security requirements. Yes, an online trading system without encryption i
simpler than an otherwise equivalent one that has it. But there's no way that it's more secure.

Choke points

Another way to improve the simplicity of your software is to funnel all security-critical operations through
of *choke points* in your system. A choke point is a narrow interface to a system that you force all traffic to g
that you can easily control it. You should avoid spreading security code throughout a system, because it bec
difficult to maintain. In addition, it's far easier to monitor what your users are doing if they're all forced to v
few small channels. This is the same theory behind sports stadiums with only a few entrances; if there were
entrances, it would be much harder to collect tickets. You'd need a lot more staff to get the same quality res

One important thing to remember about choke points: There should be no secret ways around them. For exa

stadium has an unsecured chain link fence, you can be sure that people without tickets will get in. Providing administrative features or "raw" interfaces that savvy attackers could penetrate can easily backfire. There ha of examples in which a hidden administrative backdoor could be accessed by a knowledgeable intruder, suc in the Dansie shopping cart, or a backdoor in Microsoft FrontPage, both discovered in the same month (Apr FrontPage backdoor became somewhat famous due to a hidden encryption key that read, "Netscape enginee

---

Usability

Another often-overlooked, but incredibly important aspect of simplicity is usability. Anyone who needs to u should be able to get the best security you have to offer easily, and should not be able to introduce insecurit applies to both the people who use your product and the people who have to maintain your codebase, or pro

Designers and programmers always seem to think they have an intuitive grasp of what is easy to use. Howe tests generally prove them wrong. Marginal usability might be O.K. for applications with average functiona your product might be cool enough that ease of use isn't a real concern (lucky you.) When it comes to secur usability becomes more important, because if the user interface for security isn't incredibly easy to use, ma vulnerable. Nonetheless, most people will still ignore it.

Strangely enough, there's an entire science of usability. We definitely think that all software designers shou books in this field, *The Design of Everyday Things*, by Don Norman, and *Usability Engineering*, by Jakob N Resources). This space is too small to give adequate coverage of this material. However, we can give you s as they apply to security:

1) The user will not read documentation. If the user needs to do anything special to get the full benefits of t provide, then the user is unlikely to receive those benefits. Therefore, you should provide security by defau not need to know anything about security or have to do anything in particular to be able to use your product course, security is a relative term; you will have to make some decisions as to the security requirements of y

Consider enterprise application servers that have encryption turned off by default. Generally, you can turn i menu option on an administrative tool somewhere. Even if a system administrator stops to think about secu will likely think, "They must have encryption on by default." We've seen plenty of servers that didn't.

2) Talk to users to figure out what their security requirements are. As Nielson likes to say, vice presidents a shouldn't assume that you know this information; go directly to the source. Try to provide users with more s think they need.

3) Recognize that users aren't always right. Most users aren't well-informed. They may not understand secu you should still try to anticipate their needs. It's a good idea, however, to err on the side of security. If your their needs provides more security than theirs, use yours (actually, try to provide more security than you thi theirs provides more, definitely trust them.

For example, think about a system, such as a WWeb portal, where one service you provide is stock quotes.

never think there's a good reason to secure that material. After all, stock quotes are for public consumption. good reason -- an attacker could tamper with the quotes users get. Users might then decide to buy or sell sto bogus information, and lose their shirts. Sure, you don't have to encrypt the data -- you can use a message a code (MAC) -- but most users are unlikely to anticipate this risk.

4) Users are lazy. They're so lazy that they won't actually stop to consider security, even when you throw up that says "WARNING!" in big, bright red letters. To the user, dialog boxes are an annoyance that keeps the what they want to do. For example, a lot of mobile code systems (for example, those running Active X cont browsing the WWeb with Internet Explorer) will display a dialog box that tells you who wrote the code, an really want to trust that person. Do you think anyone actually reads that stuff? Nope. Users just want to get code, and will take the path of least resistance without considering the consequences.

Principle 7: Promote privacy

Users generally consider privacy a security concern. You shouldn't do anything that could compromise the user. And you should be as diligent as possible in protecting any personal information that a user gives you heard stories about malicious hackers being able to access entire customer databases from the Web. It's also about attackers being able to hijack other users' shopping sessions, and thus gain access to private informati make every effort not to get yourself into this doghouse; you can quickly lose the respect of your customers handle privacy concerns poorly.

The privacy/usability conundrum

Maintaining privacy can often be a trade-off with usability. For example, you're better off deleting credit ca soon as you use them. That way, even if your Web site is broken into, you're not storing anything valuable hate that solution, though, because it means they have to type in their credit card data every time they want something. It makes the convenience of "one-click shopping" impossible.

The only solution here is to store the credit card information, but it's important to be very careful about this never show the user his own credit card number, in case someone manages to get unauthorized access. A co is to show a partial credit card number, with some digits blacked out. This isn't a perfect solution, either. It's keep the credit card number under lock and key; there's no reason to ever show it to the user.

We recommend asking for the issuing bank, and never revealing any part of the actual number once you ge the user wants to select a credit card, let him do it by bank. If he needs to change the number because he's o card, let him type in the information anew.

On the server side, you should encrypt the credit card number, then enter it into the database. Keep the key

card number on a different machine (this requires decrypting and encrypting on a machine other than the on
resides on). That way, if your database gets compromised, the attacker will still need to find the key, which
breaking into another machine. Anything you can do to raise the bar will help.

Secrets and lies

User privacy isn't the only kind of privacy you should consider. Malicious hackers tend to launch attacks ba
information that's easily collected from your system. Services on your machine tend to give information abo
that can help the attacker figure out how to break in. For example, the Telnet service tends to give the opera
name and version.

So how can you avoid giving out more information than necessary? First, you should be using a firewall to
unnecessary services, so that attackers can't obtain additional information from them. Second, whether you'
firewall or not (defense in depth, remember?), you should try to remove all information of this nature from
You can change the telnet login to conceal operating system specs.

In fact, why not lie about this sort of information? It can't hurt to send a potential attacker on a wild goose c
advertising your Linux box as a Solaris machine; remote Solaris exploits generally won't get someone onto
attacker might give up in frustration long before figuring out that you lied.

Leaving any sort of information around on your system can help potential attackers. Instead, you should do
your power to deter them. If you're using Rijndael as an encryption algorithm, it can't hurt to claim you're a
Twofish. Both algorithms are believed to provide similar levels of cryptographic security, so there's probab

Attackers collecting information from your environment can obtain all sorts of subtle information. For exam
idiosyncrasies of different Web server software can quickly tell you what software a system runs, even if th
modified to not report its version. Such information channels are called *covert channels*, because they're no
Usually, it's impossible to close up every covert channel in a system. This is probably the most difficult sec
identify in a system. When it comes to privacy, your best strategy is to try to identify the most likely covert
use them to your advantage by sending your potential attackers on a wild goose chase.

Next time

In our next installment, we'll finish up our list of principles with three more. First, we'll examine why you s
to be able to hide secrets in our systems, at least not without great effort. Second, we'll learn that it's import
extending trust to anyone, including ourselves. Third, we'll see that when we do need to trust, there's often g
follow the herd.

On keeping secrets, trusting others, and following the crowd

Gary McGraw (gem@cigital.com), Vice president of corporate technology, Cigital
John Viega (viega@cigital.com), Senior research associate and consultant, Cigital

01 Dec 2000

In this series, Gary and John present a set of 10 principles that can help you remove the majority of security problems from your code. Here, they finish up with the last three principles. First, they examine why you shouldn't expect to successfully hide secrets in your systems, at least not without great effort. Second, they emphasize that it's important to be wary of extending trust to anyone, including yourself. Third, they advise that when we do need to trust, it's sometimes a good idea to follow the herd.

Principle 8: It's hard to hide secrets

Security is often about keeping secrets. Users don't want their personal data leaked, so you need to keep cryptographic keys secret to avoid eavesdropping or tampering. You also want your top-secret algorithms to be protected from your competitors. These kinds of requirements are important, but far more difficult to satisfy than the average user suspects.

Many people assume that secrets in a binary are likely to stay secret, because it is too difficult to extract them. Yes, binaries are complex, but it's incredibly difficult to keep the "secrets" secret. One problem is that some people are actually quite good at reverse engineering binaries -- that is, pulling them apart, and figuring out what they do. This is why software copy protection schemes tend to be inadequate. Skilled hackers can generally circumvent any protection that a company tries to hard code into its software, and release "cracked" copies. For years, there was an escalation in the number of techniques being used; vendors would try harder to keep people from finding the secrets to "unlocking" software, and the software crackers would try harder to break the software. For the most part, the crackers won. Cracks for interesting software have been known to show up on the same day the software is officially released -- sometimes sooner.

If your software all runs server-side on your own network, you might decide that your secrets are safe. Actually, it's much harder than that to hide secrets. You shouldn't trust your own network if you can avoid it. What if some unanticipated flaw permits an intruder to steal your software? This is what happened to Id software right before they

released the first version of Quake.

Even if your network is as secure as possible, your problems may lie within. Several studies have shown that the most common threat to companies is the "insider " attack, where a disgruntled employee abuses access. Sometimes the employee isn't even disgruntled; maybe he just takes his job home, where a friend goes prodding around where he shouldn't. Also, many companies would not be able to protect their preciously-guarded software from a malicious janitor. If people are intent on getting your software through illegal means, they will probably succeed. When we point out the possibility of an inside attack to people, they often respond, "That won't happen to us; we trust our employees." If you're thinking the same thing, you should be wary. Ninety percent of the people we talk to say the same thing, yet most attacks are perpetrated by insiders. There's a huge gap here; most of the people who believe they can trust their employees must be wrong. Remember, employees may like your environment, but when it comes down to it, most of them have a business relationship with your company, not a personal one. The moral here is that it pays to be paranoid.

Sometimes people don't even need to reverse engineer software to figure out its secrets. These can often be discovered just from observing the software at work. For example, we (John Viega and Tim Hollebeek) once broke a simple cryptographic algorithm in the Netscape e-mail client simply by observing its behavior with a series of chosen inputs. (It was so easy to do that we were quoted as saying, "We didn't do this with just a pencil and paper. Lots of our notes are in pen. We didn't need to erase much.") More recently, E*Trade was the subject of a similar fiasco -- anyone could see your username and password as you logged in over the Web.

The practice of trusting a binary (or any other form of obfuscation, for that matter) to keep secrets for you is affectionately termed "security by obscurity" (see Resources). Whenever possible, you should avoid using this as your sole line of defense. That doesn't mean that there is no place for security by obscurity. Denying access to the source code definitely raises the bar for an attacker -- somewhat. Obfuscating the code to produce an obfuscated binary helps even more. These techniques require that potential attackers posses more skill than they would otherwise need to actually break your system, and that is usually a good thing. On the flip side, most systems protected this way fail to go through an adequate security audit; there is something to be said for openness, which allows you to obtain free security advice from your users.

Principle 9: Don't extend trust easily

People might more often realize their secrets are at risk if they understood that they can't trust clients that are in the hands of end users, because end users can't be trusted to use clients as they were intended. We've also urged you to be reluctant to trust your own servers, in case they get hacked (see Principle 3: Compartmentalization, referenced in Resources). This hesitancy should permeate all aspects of your security process.

For example, while off-the-shelf software can certainly help you keep your designs and implementations simple, how do you know you can trust an off-the-shelf component to

be secure? Do you really think the developers were security experts? Even if they were, do you expect them to be infallible? There have been tons of products from security vendors with gaping security holes. Many people in the security business don't actually know very much about writing secure code.

Another place where trust is generally extended far too easily is in the area of customer support. Social engineering attacks are incredibly easy to launch against unsuspecting customer support agents, who have a proclivity to trust, because it makes their jobs easier.

You should also be careful about "following the herd." Just because a particular security feature is standard doesn't mean you should provide the same poor level of protection. For example, we've often heard people opt not to encrypt sensitive data simply because their competitors weren't encrypting data. That won't be much of an excuse when customers get attacked, and then look to blame someone for being lax with security.

You should be skeptical of security vendors, too. They very often spread shady or downright false information in order to sell their products. Generally, such "snake oil" peddlers work by spreading FUD -- Fear, Uncertainty, and Doubt. Many common warning signs can help you detect quacks. One of our favorites is the advertising of "million-bit keys" for a secret-key encryption algorithm. Mathematics tells us that 256 bits is likely to be sufficient to protect messages through the lifetime of the universe -- assuming the algorithm is of high quality. People advertising more know too little about cryptography to sell worthwhile security products. Before you finish your shopping, make sure to do your research. One good place to start is the "Snake Oil" FAQ (see Resources).

You should also be reluctant to trust yourself and your organization. It's easy to be short-sighted when it comes to your own ideas and your own code. While you might like to be perfect, you should admit that you're not, and get a high-quality, objective outside perspective on what you're doing periodically.

One final point to remember is that trust is transitive. Once you give it to an entity, you implicitly extend it to anyone that entity may trust. For this reason, trusted programs should never invoke untrusted programs. You should also be careful when determining which programs to trust; programs may have hidden functionality that you do not expect. For example, in the early '90s, one of us had a UNIX account with extremely limited, menu-driven functionality. You started out in the menu when you logged in, and could only perform simple operations, such as read and compose mail and news. The menu program trusted the mail program. The mail program would call out to an external editor (in this case, the "vi" editor) when the user was composing mail. When composing mail, the user could do some vi magic to run an arbitrary command. As it turned out, it was easy to leverage this implicit, indirect trust of the vi editor to get rid of the menu system altogether, in favor of a regular old unrestricted command-line shell.

Principle 10: Trust the community

While it's not a good idea to blindly follow the herd, there is something to be said for strength in numbers. Repeated use without failure promotes trust. Public scrutiny does as well. (That is the only time this principle applies; otherwise, Principle 9 is the correct one to apply, and you should ignore this one.)

For example, in cryptography it's considered a bad idea to trust any algorithm that isn't public knowledge and hasn't been widely scrutinized. There's no real solid mathematical proof of the security of most cryptographic algorithms; they're trusted only when a bunch of smart people spend a lot of time trying to break them, and all fail to make substantial progress.

Many people find it alluring to write their own cryptographic algorithms, hoping that if these algorithms are weak, security by obscurity will serve as a safety net. Repeatedly, such hopes are dashed (for example, with the Netscape and E*Trade breaks mentioned above). The argument generally goes that a secret algorithm is better than a publicly-known one. We've already discussed how you should expect your algorithm not to stay secret for very long. For example, the RC4 encryption algorithm was supposed to be a trade secret of RSA Data Security. However, it was reverse engineered and posted on the Internet anonymously soon after its introduction.

In fact, cryptographers design their algorithms so that knowledge of the algorithm is unimportant to security. Good cryptographic algorithms work because you keep a small secret called a "key," not because the algorithm is secret. That is, the only thing you need to keep private is the key. If you can do that, and the algorithm is actually good (and the key is long enough), then even an attacker who's intimately familiar with the algorithm will be unable to attack you.

Similarly, it's far better to trust security libraries that have been widely used, and widely scrutinized. Sure, they might contain bugs that haven't been found -- but at least you get to leverage the experience of others.

This principle only applies if you have reason to believe that the community is doing its part to promote the security of components you want to use. One common misconception is the belief that "open source" software is highly likely to be secure, because source availability will lead to people performing security audits. There's strong evidence to suggest that source availability doesn't provide the strong incentive for people to review source code, even though many people would like to believe that incentive exists. For example, many security bugs in widely-used, free software programs went unnoticed for years. In the case of the most popular FTP server around (WU-FTPD), several security bugs went unnoticed for more than a decade!

Conclusion

In the past five installments, we've talked about a set of general principles that we think can help you avoid most security problems. The principles are:

1. Identify and reinforce the weakest link.

2. Provide defense in depth, which means you should manage software risk by providing redundant security solutions. Usually, one level of redundancy is worthwhile; whether you need more depends on your particular project.
3. Secure failure: Make sure that if the system could possibly fail, it will fail in a secure manner.
4. Least privilege: Do not give out more privileges than necessary, and do not extend privileges longer than necessary.
5. Compartmentalization: Try to keep failures in one part of a system from having an impact on the rest of the system.
6. Keep it simple.
7. Privacy: Don't give out any unnecessary information.
8. It's hard to hide secrets.
9. Don't extend trust easily.
10. Trust the community.

Remember that it's important to apply good software risk management techniques when using these principles. You should only apply principles when such a strategy tells you that it is cost effective to do so. Also, these principles can sometimes work to contradictory ends, so such a strategy is even more crucial. Finally, we certainly don't expect that this set of principles will cover every situation you might encounter, but it should do a good job in general.