Design Principles

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-09-19 L4 / D/P, L[5]

As the recognition of security as a key dimension of high-quality software development has grown, the understanding of and ability to craft secure software has become a more common expectation of software developers. The challenge is in the learning curve. Most developers don't have the benefit of years and years of lessons learned that an expert in software security can call on. In an effort to bridge this gap, the Principles content area, along with the Guidelines and Coding Rules content areas, presents a set of practices derived from real-world experience that can help guide software developers in building more secure software.

Jerome Saltzer and Michael Schroeder were the first researchers to correlate and aggregate high-level security principles in the context of protection mechanisms [Saltzer 75]. Their work provides the foundation needed for designing and implementing secure software systems. Principles define effective practices that are applicable primarily to architecture-level software decisions and are recommended regardless of the platform or language of the software. As with many architectural decisions, the principles, which do not necessarily guarantee security, at times may exist in opposition to each other, so appropriate tradeoffs must be made. Software developers, whether they are crafting new software or evaluating and assessing existing software, should always apply these design principles as a guide and yardstick for making their software more secure.

The Principles content area presents several principles, many from Saltzer and Schroeder's original work and a couple of others from other thought leaders in the space. The filter applied to decide what is a principle and what is not is fairly narrow, recognizing that such lasting principles do not come along every day and that the term has been overused recently to define many things, causing confusion. Each principle consists of a brief description outlining the basic concept of the principle and then a set of more detailed descriptions in the form of block quotes from recognized thought leader publications describing their perspective on that particular principle. Rather than instigating conflict by acting as self-appointed arbiters in defining the one true interpretation of each principle, we decided to present readers with the different points of view available and allow them to make their own interpretations based on their personal trust filters. In doing this, editorial comment and explanatory prose has been kept to a minimum by design. It is our hope that readers of the principles, both expert and novice alike, will contribute to this explanatory discussion through the collaboration channels available on Build Security In. Eventually, these principles will likely be enhanced with content from those discussions.

The Principles for Software Security

- Securing the Weakest Link[7]
- Defense in Depth[8]
- Failing Securely[9]
- Least Privilege[10]
- Separation of Privilege[11]
- Economy of Mechanism[12]
- Least Common Mechanism[13]
- Reluctance to Trust[14]

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)

- Never Assuming that your Secrets are Safe[15]
- Complete Mediation[16]
- Psychological Acceptability[17]
- Promoting Privacy[18]

References

[Saltzer 75] Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems," 1278-1308. *Proceedings of the IEEE 63,* 9 (September 1975).

Cigital, Inc. Copyright

The Build Security In (BSI) portal is sponsored by the U.S. Department of Homeland Security (DHS), National Cyber Security Division. The Software Engineering Institute (SEI) develops and operates BSI. DHS funding supports the publishing of all site content.

1. mailto:copyright@cigital.com

Securing the Weakest Link

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-09-19 L4 / D/P, L[5]

Attackers are more likely to attack a weak spot in a software system than to penetrate a heavily fortified component. For example, some cryptographic algorithms can take many years to break, so attackers are not likely to attack encrypted information communicated in a network. Instead, the endpoints of communication (e.g., servers) may be much easier to attack. Knowing when the weak spots of a software application have been fortified can indicate to a software vendor whether the application is secure enough to be released.

Detailed Description Excerpts

According to Viega and McGraw [Viega 02] in Chapter 5, "Guiding Principles for Software Security," in "Principle 1: Secure the Weakest Link" from pages 93-96:[9]

> Security practitioners often point out that security is a chain; and just as a chain is only as strong as the weakest link, a software security system is only as secure as its weakest component. Bad guys will attack the weakest parts of your system because they are the parts most likely to be easily broken. (Often, the weakest part of your system will be administrators, users or tech support people who fall prey to social engineering.)
>
> It's probably no surprise to you that attackers tend to go after low-hanging fruit. If a malicious hacker targets your system for whatever reason, they're going to follow the path of least resistance. That means they'll try to attack the parts of the system that look the weakest, and not the parts that look the strongest. (Of course even if they spend an equal effort on all parts of your system, they're far more likely to find exploitable problems in the parts of your system most in need of help.)
>
> A similar sort of logic pervades the physical security world. 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, because banks tend to have much stronger security precautions. Convenience stores are thus a much easier target. Of course the payoff for successfully robbing a convenience store is much lower than knocking off a bank; but it is probably a lot easier to get away from the convenience store crime scene.
>
> So to stretch our analogy a bit, you want to look for and better defend the convenience stores in your software system.
>
> Consider cryptography. Cryptography is seldom the weakest part of a software system. Even if a system uses SSL-1 with 512-bit RSA keys and 40-bit RC4 keys (which is, by the way, considered an incredibly weak system all around) an attacker can probably find much easier ways to break the system than attacking the crypto. Even though this system is definitely breakable through a concerted crypto attack, successfully carrying out the attack requires a large computational effort and some knowledge of cryptography.
>
> Let's say the bad guy in question wants access to secret data being sent from point A to point B over the network (traffic protected by SSL-1). A clever attacker will target one of the endpoints, try to find a flaw like a buffer overflow, and then look at the data before it gets encrypted, or after it gets decrypted. Attacking the data while encrypted is just too much work. All the cryptography in the world can't help you if there's an exploitable buffer overflow, and buffer overflows abound in code written in C.

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)
9. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.

For this reason, while cryptographic key lengths can certainly have an impact on the security of a system, they aren't all that important in most systems, where there exist much bigger and more obvious targets.

For similar reasons, attackers don't attack a firewall unless there's a well-known vulnerability in the firewall itself (something all too common, unfortunately). Instead, they'll try to break the applications that are visible through the firewall, since these applications tend to be much easier targets. New development tricks and protocols like SOAP, a system for tunneling traffic through port 80, make our observation even more relevant. It's not about the firewall; it's about what is listening on the other side of the firewall.

Identifying the weakest component of a system falls directly out of a good risk analysis. Given good risk analysis data, addressing the most serious risk first, instead of a risk that may be easiest to mitigate, is always prudent. Security resources should be doled out according to risk. Deal with one or two major problems, and move on to the remaining ones in order of severity.

Of course, this strategy can be applied forever, since 100% security is never attainable. There is a clear need for some stopping point. It is okay to stop addressing risks when all components appear to be within the threshold of acceptable risk. The notion of acceptability depends on the business proposition, of course.

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, an attack in which a bad guy uses social manipulation to break into a system. In a typical scenario, a service center will get a call from a sincere sounding user, who will talk the service professional out of a password that should never be given away. This sort of attack is easy to carry out, because customer service representatives don't like to deal with stress. If they are faced with a customer who seems to be really mad about not being able to get into their account, they may not want to aggravate the situation by asking questions to authenticate the remote user. They will instead be tempted just to change the password to something new and be done with it.

To do this right, the representative should verify that the caller is in fact the user in question who needs a password change. Even if they do ask questions to authenticate the person on the other end of the phone, what are they going to ask? Birthdate? Social Security number? Mother's maiden name? All of that information is easy for a bad guy to get if they know their 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 (remember, less functionality means less security exposure). For example, you might choose to make it impossible for a user to change a password. If a user forgets their password, then the solution is to create another account. Of course, that particular example is not always an appropriate solution, since it is a real inconvenience for users. Relying on caller ID is a better scheme, but that doesn't always work either. That is, caller ID isn't available everywhere. Moreover, perhaps the user is on the road, or the attacker can convince a customer service representative that they are the user on the road.

The following somewhat elaborate scheme presents a reasonable solution. Before deploying the system, a large list of questions is composed (say, no fewer than 400 questions). Each question should be generic enough that any one person should be able to answer it. However, the answer to any single question should be pretty difficult to guess (unless you are the right person). When the user creates an account, we select 20 questions from the list, and ask the user to answer 6 of them that the user has answers for, and is most likely to give the same answer if asked again in two years.

Here are some sample questions:

- What is the name of the celebrity you think you most resemble, and the one you would most like to resemble?
- What was your most satisfying accomplishment in your high school years?
- List the first names of any significant others you had in high school.

- Whose birth was the first birth that was significant to you, be it a person or animal?
- Who is the person in whom you were most interested to whom you never expressed your interest (your biggest secret crush)?

When someone forgets their password and calls technical support, technical support refers the user to a web page (that's all they are given the power to do). The user is provided with three questions from the list of six, and must answer two correctly. If they answer 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 a user to authenticate in this way a small handful of times (say, three).

The result of this scheme is that users can get done what they need to get done when they forget their passwords, but tech support is protected from social engineering attacks. We're thus fixing the weakest link in a system.

All of our asides aside, good security practice dictates an approach that identifies and strengthens weak links until an acceptable level of risk is achieved.

According to Schneier [Schneier 00] in "Security Processes":

> **Secure the Weakest Link.** Spend your security budget securing the biggest problems and the largest vulnerabilities. Too often, computer security measures are like planting an enormous stake in the ground and hoping the enemy runs right into it. Try to build a broad palisade.

Further Reading

Crafting malicious input is one method of circumventing strong encryption or user authentication. For example, an attacker need only inject a simple SQL command in a web form to obtain a list of usernames that are encrypted in a database. The failure to sanitize user input that contains SQL commands represents a weak link that attackers often prey on. Other examples of malformed input include the injection of shell commands that are executed with root privileges and cross-site scripting attacks, in which script code is injected in HTML pages that can reveal a user's cookie information. Attackers don't stop at text entry fields; they can also manipulate protocols (e.g., HTTP GET requests and payload information) to wreak havoc on a victim machine. Preventing users from being able to enter arbitrary data into an application is crucial for providing good security.

A *black list* [Hoglund 04] can be created that contains possible forms of malicious input for a data entry field that developers can check against in their code. However, determining all variations of unsafe input is infeasible because of the sheer number of possible exploits an attacker can employ. Instead, developers may code according to what a *white list* [Hoglund 04] defines as well-formed input for a given input field. Good requirements engineering makes it possible to know exactly what input is expected so the proper security checks can be put in place. Securing the code that interfaces with the user will indubitably decrease the likelihood of a successful attack.

References

[Hoglund 04]	Hoglund, Greg & McGraw, Gary. *Exploiting Software: How to Break Code.* Boston, MA: Addison-Wesley, 2004.
[Schneier 00]	Schneier, Bruce. "The Process of Security[14]." *Information Security Magazine,* April, 2000.
[Viega 02]	Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.

Cigital, Inc. Copyright

The Build Security In (BSI) portal is sponsored by the U.S. Department of Homeland Security (DHS), National Cyber Security Division. The Software Engineering Institute (SEI) develops and operates BSI. DHS funding supports the publishing of all site content.

1. mailto:copyright@cigital.com

Defense in Depth

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-09-13 L4 / D/P, L[5]

Layering security defenses in an application can reduce the chance of a successful attack. Incorporating redundant security mechanisms requires an attacker to circumvent each mechanism to gain access to a digital asset. For example, a software system with authentication checks may prevent an attacker that has subverted a firewall. Defending an application with multiple layers can prevent a single point of failure that compromises the security of the application.

Detailed Description Excerpts

According to Viega and McGraw [Viega 02] in Chapter 5, "Guiding Principles for Software Security," in "Principle 2: Practice Defense in Depth" from pages 96-97:[9]

> The idea behind defense in depth is to manage risk with diverse defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense will hopefully 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.[10]
>
> Let's go back to our example of bank security. Why is the typical bank more secure than the typical convenience store? Because there are many redundant security measures protecting the bank, and the more measures there are, the more secure the place is.
>
> Security cameras alone are a deterrent for some. But if people don't care about the cameras, then a security guard is there to physically defend the bank with a gun. Two security guards provide even more protection. But if both security 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 from the robbers. Of course if the robbers happen to kick in the doors, or guess the code for the door, at least they can only get at the teller registers, since we have a vault protecting the really valuable stuff. Hopefully, the vault is protected by several locks, and cannot be opened without two individuals who are rarely at the bank at the same time. And as for the teller registers, they can be protected by having dye-emitting bills stored at the bottom, for distribution during a robbery.
>
> Of course, having all these security measures does not ensure that our bank will never be successfully robbed. 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 would.
>
> The defense in depth principle may seem somewhat contradictory to the "secure the weakest link" principle, since we are essentially saying that defenses taken as a whole can be stronger than the weakest link. However, there is no contradiction; the principle "secure 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 sum protection offered is far greater than the protection offered by any single component.

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)
9. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.
10. MacLennan, Bruce. *Principles of Programming Languages*. Holt,Rinehart and Winston, 1987.

A good real-world example where defense-in-depth can be useful, but is rarely applied, is in the protection of data that travel between various server components in enterprise systems. Most companies will throw up a corporate-wide firewall to keep intruders out. Then they'll assume that the firewall is good enough, and let their application server talk to their database in the clear. Assuming that the data in question are important, what happens if an attacker manages to penetrate the firewall? If the data are also encrypted, then the attacker won't be able to get at them without breaking the encryption, or (more likely) breaking onto one of the servers that stores the data in an unencrypted form. If we throw up another firewall, just around the application this time, then we can protect ourselves from people who can get inside the corporate firewall. Now they'd have to find a flaw in some service that our application's sub-network explicitly exposes, something we're in a good position to control.

Defense in depth is especially powerful when each layer works in concert with the others.

According to Howard and LeBlanc [Howard 02[11]] in Chapter 3, "Security Principles to Live By," in "Use Defense in Depth," from pages 59-60:

Defense in depth is a straightforward principle: imagine your application is the last component standing and every defensive mechanism protecting you has been destroyed. Now you must protect yourself. For example, if you expect a firewall to protect you, build the system as though the firewall has been compromised.

Unfortunately, a great deal of software is designed and written in a way that leads to total compromise when a firewall is breached. This is not good enough today. Just because some defensive mechanism has been compromised doesn't give you the right to concede defeat. This is the essence of defense in depth: at some stage you have to defend yourself. Don't rely on other systems to protect you. Put up a fight because software fails, hardware fails, and people fail. People build software, people are flawed, and therefore software is flawed. You must assume that errors will occur that will lead to security vulnerabilities. That means the single layer of defense in front of you will probably be compromised, so what are your plans if it is defeated? Defense in depth helps reduce the likelihood of a single point of failure in the system.

Important: Always be prepared to defend your application from attack because the security features defending it might be annihilated. Never give up.

Example

Let's quickly revisit the castle example from the first chapter. This time, your users are the noble family of a castle in the 1500s, and you are the captain of the army. The bad guys are coming, and you run to the lord of the castle to inform him of the encroaching army and of your faith in your archers, the castle walls, and the castle's moat. The lord is pleased. Two hours later you ask for an audience with the lord and inform him that the marauders have broken the defenses and are inside the outer wall. He asks how you plan to further defend the castle. You answer that you plan to surrender because the bad guys are inside the castle walls. A response like yours doesn't get you far in the armed forces. You don't give up--you keep fighting until all is lost or you're told to stop fighting.

Here's another example, one that's a little more modern. Take a look at a bank. When was the last time you entered a bank to see a bank teller sitting on the floor in a huge room next to a massive pile of money. Never! To get to the big money in a bank requires that you get to the bank vault, which requires that you go through multiple layers of defense. Here are some examples of the defensive layers:

- There is often a guard at the bank's entrance.
- Some banks have time-release doors. As you enter the bank, you walk into a bulletproof glass capsule. The door you entered closes, and after a few seconds the glass door to the

11. #dsy347-BSI_refs

bank opens. This means you cannot rush in and rush out. In fact, a teller can lock the doors remotely, trapping a thief as he attempts to exit.

- There are guards inside the bank.
- Numerous closed-circuit cameras monitor the movements of every one in every corner of the bank.
- Tellers do not have access to the vault. (This is an example of least privilege, which is covered next.)
- The vault itself has multiple layers of defense, such as:
  - It opens only at certain controlled times.
  - It's made of very thick metal.
  - Multiple compartments in the vault require other access means.

According to NIST [NIST 01] in Section 3.3, "IT Security Principles," from page 9:

> Implement layered security (ensure no single point of vulnerability). Security designs should consider a layered approach to address or protect against a specific threat or to reduce a vulnerability. For example, the use of a packet-filtering router in conjunction with an application gateway and an intrusion detection system combine to increase the work-factor an attacker must expend to successfully attack the system. Adding good password controls and adequate user training improves the system's security posture even more.
>
> The need for layered protections is especially important when commercial-off-the-shelf (COTS) products are used. Practical experience has shown that the current state-of-the-art for security quality in COTS products does not provide a high degree of protection against sophisticated attacks. It is possible to help mitigate this situation by placing several controls in series, requiring additional work by attackers to accomplish their goals.

According to Schneier [Schneier 00] in "Security Processes":

> Provide Defense in Depth.
>
> Don't rely on single solutions. Use multiple complementary security products, so that a failure in one does not mean total insecurity. This might mean a firewall, an intrusion detection system and strong authentication on important servers.

References

[Howard 02]	Howard, Michael & LeBlanc, David. *Writing Secure Code, 2nd ed*. Redmond, WA: Microsoft Press, 2002.
[NIST 01]	*Engineering Principles for Information Technology Security*. Special Publication 800-27. US Department of Commerce, National Institute of Standards and Technology, 2001.
[Schneier 00]	Schneier, Bruce. "The Process of Security[13]." *Information Security Magazine,* April, 2000.
[Viega 02]	Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.

Cigital, Inc. Copyright

The Build Security In (BSI) portal is sponsored by the U.S. Department of Homeland Security (DHS), National Cyber Security Division. The Software Engineering Institute (SEI) develops and operates BSI. DHS funding supports the publishing of all site content.

1. mailto:copyright@cigital.com

Failing Securely

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-12-05 L4 / D/P, L[5]

When a system fails, it should do so securely. This typically involves several things: secure defaults (default is to deny access); on failure undo changes and restore to a secure state; always check return values for failure; and in conditional code/filters make sure that there is a default case that does the right thing. The confidentiality and integrity of a system should remain even though availability has been lost. Attackers must not be permitted to gain access rights to privileged objects during a failure that are normally inaccessible. Upon failing, a system that reveals sensitive information about the failure to potential attackers could supply additional knowledge for creating an attack. Determine what may occur when a system fails and be sure it does not threaten the system.

Detailed Description Excerpts

According to [Saltzer 75] in "Basic Principles of Information Protection" on page 8:

> Fail-safe defaults: Base access decisions on permission rather than exclusion. This principle, suggested by E. Glaser in 1965,[9] means that the default situation is lack of access, and the protection scheme identifies conditions under which access is permitted. The alternative, in which mechanisms attempt to identify conditions under which access should be refused, presents the wrong psychological base for secure system design. A conservative design must be based on arguments why objects should be accessible, rather than why they should not. In a large system some objects will be inadequately considered, so a default of lack of permission is safer. A design or implementation mistake in a mechanism that gives explicit permission tends to fail by refusing permission, a safe situation, since it will be quickly detected. On the other hand, a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure which may go unnoticed in normal use. This principle applies both to the outward appearance of the protection mechanism and to its underlying implementation.

According to Viega and McGraw [Viega 02] in Chapter 5, "Guiding Principles for Software Security," in "Principle 3: Fail Securely" on pages 97-100:[10]

> Any sufficiently complex system will have failure modes. Failure is unavoidable and should be planned for. What is avoidable are security problems related to failure. The problem is that when many systems fail in any way, they exhibit 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. Then they can go to town.
>
> The best real world example we know 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 calls up 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 context of your recent

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)
9. In this material we have attempted to identify original sources whenever possible. Many of the seminal ideas, however, were widely spread by word of mouth or internal memorandum rather than by journal publication, and historical accuracy is sometimes difficult to obtain. In addition, some ideas related to protection were originally conceived in other contexts. In such cases, we have attempted to credit the person who first noticed their applicability to protection in computer systems, rather than the original inventor.
10. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.

purchases and compares the patterns to the overall spending trends. If their engine senses anything suspicious, the transaction is denied. (Sometimes the trend analysis is performed off line and the owner of a suspect card gets a call later.)

This scheme appears to be remarkably impressive from a security point of view, that is until you note what happens when something goes wrong. What happens if the line to the credit card company is down? Is the vendor required to say, "I'm sorry, our phone line is down"? No. The credit card company still gives out manual machines that take an imprint of your card, which the vendor can send to the credit card company for reimbursement later. An attacker need only cut the phone line before ducking into a 7-11.

There used to be some security in the manual system, but it's largely gone now. Before computer networks, a customer was supposed to be asked for identification to make sure the card matched a license or some other id. Now, people rarely get asked for identification when making purchases; we rely on the computer instead. The credit card company can live with authenticating a purchase or two before a card is reported stolen; it's an acceptable risk. Another precaution was that if your number appeared on a periodically updated paper list of bad cards in the area, the card would be confiscated. Also, the vendor would 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 need to come back into play. In practice, they tend not to, though. Failure is fortunately so uncommon in credit card systems that there is no justification for asking vendors to remember complex procedures when it does happen. That means when the system fails, the behavior of the system is less secure than typical behavior. How difficult is it to make the system fail?

Why do credit card companies use such a brain dead fallback scheme? The answer is that the credit card companies are good at risk management. They can eat 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, since the amount of fraud is relatively low (there are a lot of factors considered in this decision, including business costs and public relations issues).

Plenty of other examples are to be found in the digital world. Often, the insecure failure problem occurs because of a desire to support legacy versions of software that were not secure. For example, let's say that the original version of your software was broken, and did not use encryption to protect sensitive data at all. Now you want to fix the problem, but you have 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 don't use the new protocols. You'd like to force old users to upgrade, but you didn't plan 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 get from the other, 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 think each of the others is an old client by tampering with data as it traverses the network. (A form of man-in-the-middle-attack.) Worse yet, there's no way to get rid of the problem while still supporting full (two-way) backwards compatibility.

A good solution to this problem is to design in a forced upgrade path from the very beginning. One way is to make sure that the client can detect that the server is no longer supporting it. If the client can securely retrieve patches, it will be forced to do so. Otherwise, it tells the user that a new copy must be obtained manually. Unfortunately, it's important to have this sort of solution in place from the very beginning. That is, unless you don't mind alienating early adopters.

We discussed a problem similar to the one in Chapter 3 [in *Building Secure Software*], which exists in most implementations of Java's Remote Method Invocation (RMI). When a client and server wish to communicate over RMI, but the server wants to use SSL or some other protocol other than "no encryption", the client may not support the protocol the server would like to use. When that's the case, the client will generally download the proper socket implementation from the server at runtime. This constitutes a big security hole, because the server has yet to be authenticated at the time that the

encryption interface is downloaded. That means an attacker can pretend to be the server, installing a malicious socket implementation on each client, even when the client already had 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 when it should not be extended.

If your software has to fail, make sure it does so securely!

We include two relevant principles from Howard and LeBlanc [Howard 02] in Chapter 3, "Security Principles to Live By," in "Plan on Failure" on page 64:[11]

As I've mentioned, stuff fails and stuff breaks. In the case of mechanical equipment, the cause might be wear and tear, and in the case of software and hardware, it might be bugs in the system. Bugs happen—plan on them occurring. Make security contingency plans. What happens if the firewall is breached? What happens if the Web site is defaced? What happens if the application is compromised? The wrong answer is, "It'll never happen!" It's like having an escape plan in case of fire—you hope to never have to put the strategy into practice, but if you do you have a better chance of getting out alive.

Tip: Death, taxes, and computer system failure are all inevitable to some degree. Plan for the event.

And this from Chapter 3, "Security Principles to Live By," in "Fail to a Secure Mode" on pages 64-66:[12]

So, what happens when you do fail? You can fail securely or insecurely. Failing to a secure mode means the application has not disclosed any data that would not be disclosed ordinarily, that the data still cannot be tampered with, and so on. Or you can fail insecurely such that the application discloses more than it should or its data can be tampered with (or worse). The former is the only proposition worth considering—if an attacker knows that he can make your code fail, he can bypass the security mechanisms because your failure mode is insecure.

Also, when you fail, do not issue huge swaths of information explaining why the error occurred. Give the user a little bit of information, enough so that the user knows the request failed, and log the details to some secure log file, such as the Windows event log.

Important: The golden rule when failing securely is to deny by default and allow only once you have verified the conditions to allow.

Example

For a microview of insecure failing, look at the following (pseudo) code and see whether you can work out the security flaw:

```
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == ERROR_ACCESS_DENIED) {
  // Security check failed.
  // Inform user that access is denied.
} else {
  // Security check OK.
}
```

At first glance, this code looks fine, but what happens if IsAccessAllowed fails? For example, what happens if the system runs out of memory, or object handles, when this function is called? The user can execute the privileged task because the function might return an error such as ERROR NOT ENOUGH MEMORY.

The correct way to write this code is as follows:

```
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == NO_ERROR) {
  // Secure check OK.
  // Perform task.
```

11. From *Writing Secure Code*, *Second Edition* (0-7356-1722-8) by Microsoft Press. All rights reserved.
12. From *Writing Secure Code*, *Second Edition* (0-7356-1722-8) by Microsoft Press. All rights reserved.

```
} else {
  // Security check failed.
  // Inform user that access is denied.
}
```

In this case, if the call to IsAccessAllowed fails for any reason, the user is denied access to the privileged operation.

A list of access rules on a firewall is another example. If a packet does not match a given set of rules, the packet should not be allowed to traverse the firewall; instead, it should be discarded. Otherwise, you can be sure there's a corner case you haven't considered that would allow a malicious packet, or a series of such packets, to pass through the firewall. The administrator should configure firewalls to allow only the packet types deemed acceptable though, and everything else should be rejected.

Another scenario, covered in detail in Chapter 10 [of *Writing Secure Code*], "All Input is Evil!" is to filter user input looking for potentially malicious input and rejecting the input if it appears to contain malevolent characters. A potential security vulnerability exists if an attacker can create input that your filter does not catch. Therefore, you should determine what is valid 'input and reject all other input."

According to Bishop [Bishop 03] in Chapter 13, "Design Principles," in Section 13.2.2, "Principle of Fail-Safe Defaults," on page 344:[13]

This principle restricts how privileges are initialized when a subject or object is created.

Definition 13-2. The Principle of Fail-Safe Defaults states that, unless a subject is given explicit access to an object, it should be denied access to that object.

This principle requires that the default access to an object is none. Whenever access, privileges, or some security-related attribute is not explicitly granted, it should be denied. Further, if the subject is unable to complete its action or task, before the subject terminates, it should undo those changes it made to the security state of the system. This way, even if the program fails, the system is still safe.

Example 1

If the mail server is unable to create a file in the spool directory, it should close the network connection, issue an error message, and stop. It should not try to store the message elsewhere, nor expand its privileges to save the message in another location because an attacker could use that ability to overwrite other files or fill up other disks (a denial of service attack). The protections on the mail spool directory itself should allow create and write access to only the mail server, and read and delete access to only the local server. No other user should have access to the directory.

In practice, most systems will allow an administrator access to the mail spool directory. By the principle of least privilege, that administrator should only be able to access the subjects and objects involved in mail queueing and delivery. As we saw, this minimizes the threats if that administrator's account is compromised. The mail system can be damaged or destroyed, but nothing else can be.

According to NIST [NIST 01] in Section 3.3, "IT Security Principles," on page 10:

Design and operate an IT system to limit vulnerability and to be resilient in response.

Information systems should be resistant to attack, should limit damage, and should recover rapidly when attacks do occur. The principle suggested here recognizes the need for adequate protection technologies at all levels to ensure that any potential cyber attack will be countered effectively. There are vulnerabilities that cannot be fixed, those that have not yet been fixed, those that are not known, and those that could be fixed but are not (e.g., risky services allowed through firewalls) to allow increased operational capabilities. In addition to achieving a secure initial state, secure systems should have a well-defined status after failure, either to a secure failure state or via a recovery procedure to

13. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.

a known secure state. Organizations should establish detect and respond capabilities, manage single points of failure in their systems, and implement a reporting strategy.

According to Schneier [Schneier 00] in "Security Processes":

> Fail Securely.
>
> Design your networks so that when products fail, they fail in a secure manner. When an ATM fails, it shuts down; it doesn't spew money out its slot.

"What Goes Wrong"

According to McGraw and Viega [McGraw 03]:[16]

> A flaw in this DNS-spoofing detector dulled its paranoia.
>
> If your software doesn't fail safely, you're in trouble. An example reported in August 2001 was found in versions 3.2 to 4.3 of the FreeBSD operating system, where the tcp_wrappers PARANOID hostname checking does not work properly. A flawed check for a numeric result during reverse DNS lookup caused the broken code in tcp_wrappers to skip some of its sanity checking for DNS results, allowing access to an attacker impersonating a trusted host (see the full explanation on the Bugtraq list[17]).

References

[Bishop 03]	Bishop, Matt. *Computer Security: Art and Science*. Boston, MA: Addison-Wesley, 2003.
[Howard 02]	Howard, Michael & LeBlanc, David. *Writing Secure Code*. 2nd. Redmond, WA: Microsoft Press, 2002.
[McGraw 03]	McGraw, Gary & Viega, John. "Keep It Simple." *Software Development.* CMP Media LLC, May, 2003.
[NIST 01]	*Engineering Principles for Information Technology Security*. Special Publication 800-27. US Department of Commerce, National Institute of Standards and Technology, 2001.
[Saltzer 75]	Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems," 1278-1308. *Proceedings of the IEEE 63*, 9 (September 1975).
[Schneier 00]	Schneier, Bruce. "The Process of Security[19]." *Information Security Magazine,* April, 2000.
[Viega 02]	Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.

Cigital, Inc. Copyright

16. All rights reserved. It is reprinted with permission from CMP Media LLC.
17. http://online.securityfocus.com/advisories/3515

The Build Security In (BSI) portal is sponsored by the U.S. Department of Homeland Security (DHS), National Cyber Security Division. The Software Engineering Institute (SEI) develops and operates BSI. DHS funding supports the publishing of all site content.

1. mailto:copyright@cigital.com

Least Privilege

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-09-14 L4 / D/P, L[5]

Only the minimum necessary rights should be assigned to a subject that requests access to a resource and should be in effect for the shortest duration necessary (remember to relinquish privileges). Granting permissions to a user beyond the scope of the necessary rights of an action can allow that user to obtain or change information in unwanted ways. Therefore, careful delegation of access rights can limit attackers from damaging a system.

Detailed Description Excerpts

According to Saltzer and Schroeder [Saltzer 75] in "Basic Principles of Information Protection," page 9:

> Least privilege: Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur. Thus, if a question arises related to misuse of a privilege, the number of programs that must be audited is minimized. Put another way, if a mechanism can provide "firewalls," the principle of least privilege provides a rationale for where to install the firewalls. The military security rule of "need-to-know" is an example of this principle.

According to Bishop [Bishop 03] in Chapter 13, "Design Principles," Section 13.2.1, "Principle of Least Privilege," pages 343-344:[9]

> This principle restricts how privileges are granted.
>
> Definition 13-1. The Principle of Least Privilege states that a subject should be given only those privileges needed for it to complete its task.
>
> If a subject does not need an access right, the subject should not have that right. Further, the function of the subject (as opposed to its identity) should control the assignment of rights. If a specific action requires that a subject's access rights be augmented, those extra rights should be relinquished immediately upon completion of the action. This is the analogue of the "need to know" rule: if the subject does not need access to an object to perform its task, it should not have the right to access that object. More precisely, if a subject needs to append to an object, but not to alter the information already contained in the object, it should be given append rights and not write rights.
>
> In practice, most systems do not have the needed granularity of privileges and permissions to apply this principle precisely. The designers of security mechanisms then apply this principle as best they can. In such systems, the consequences of security problems are often more severe than the consequences on systems which adhere to this principle.
>
> This principle requires that processes should be confined to as small a protection domain as possible.
>
> Example 1

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)
9. This excerpt is from the book *Computer Security: Art and Science*, written by Matt Bishop, ISBN 0-201-44099-7, copyright 2003. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.

The UNIX operating system does not apply access controls to the user root. That user can terminate any process and read, write, or delete any file. Thus, users who create back-ups can also delete files. The administrator account on Windows has the same powers.

Example 2

A mail server accepts mail from the Internet, and copies the messages into a spool directory; a local server will complete delivery. It needs rights to access the appropriate network port, to create files in the spool directory, and to alter those files (so it can copy the message into the file, rewrite the delivery address if needed, and add the appropriate "Received" lines). It should surrender the right to access the file as soon as it has completed writing the file into the spool directory, because it does not need to access that file again. The server should not be able to access any user's files, or any files other than its own configuration files.

According to Viega and McGraw [Viega 02] in Chapter 5, "Guiding Principles for Software Security," in "Principle 4: Follow the Principle of Least Privilege" from pages 100-103:[10]

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. (This principle was introduced by Saltzer and Schroeder.[11])

When you give out access to parts of a system, there is always some risk that the privileges associated with that access will be abused.

This problem is starting to become common in security policies that ship with products intended to run in a restricted environment. Some vendors offer applications that work as Java applets. Applets usually constitute mobile code, which a web browser treats with suspicion by default. Such code is run in a sandbox, where the behavior of the applet is restricted based on a security policy that a user sets. Vendors rarely practice the principle of least privilege when they suggest a policy to use with their code, because doing so would take a lot of effort on their part. It's far easier to just ship a policy that says, "let my code do anything at all." People will generally install vendor-supplied security policies, maybe because they trust the vendor, or maybe because it's too much hassle to figure out what security policy does the best job of minimizing the privileges that must be granted to the vendor's application.

Laziness often works against the principle of least privilege. Don't let that happen in your code.

Example 1

For example, let's say you were to go on vacation, and give a friend the key to your home, just to feed pets, collect mail, etc. While you may trust a 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 really no need to put yourself at risk by giving more access than necessary. For example, if you don't have pets, but only needed a friend to occasionally pick up our 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 is not being supervised by you constitutes a window of time in which you are vulnerable to an attack. You want to keep such windows of vulnerability as short as possible, in order to minimize your risks.

---

10. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.
11. Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems." *Proceedings of the IEEE 63*, 9 (September 1975): 1278-1308.

Example 2

Another good real-world example appears in the security clearance system of the U.S. government; in particular with the notion of "need to know". If you have clearance to see any secret document whatsoever, you still can't demand to see any secret document that you know exists. If you could, it would be very easy to abuse the security clearance level. Instead, people are only allowed to access documents that are relevant to whatever task they are supposed to perform.

Example 3

Some of the most famous violations of the principle of least privilege exist in UNIX systems. For example, in UNIX systems, root privileges are necessary to bind a program to a port number less than 1024. For example, 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 relinquishes root privileges as soon as possible, and will let the operating system know that it should never require those privileges again in this execution (see Chapter 8 [in *Building Secure Software*] for a discussion of privileges). One large problem with many e-mail servers is that they don't give up their root permissions once they grab the mail port (Sendmail is a classic example). Therefore, if someone ever finds a way to trick such a mail server into doing something nefarious, it will be able to get root. So if a malicious attacker were to find a suitable stack overflow in Sendmail (see Chapter 7 [in *Building Secure Software*]), that overflow could be used to trick the program into running arbitrary code as root. Given root permission, anything valid the attacker tries will succeed. The problem of relinquishing privilege is especially bad in Java, since there is no operating-system independent way to give up permissions.

Example 4

Another common scenario involves a programmer who may wish to access some sort of data object, but only needs to read from the object. Let's say the programmer actually requests more privileges than necessary, for whatever reason. Programmers do this to make life easier. For example, one might say, "Someday I might need to write to this object, and it would suck 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, since that's easiest.

According to Howard and LeBlanc [Howard 02] in Chapter 3, "Security Principles to Live By," in "Use Least Privilege" from pages 60-61:

> All applications should execute with the least privilege to get the job done and no more. I often analyze products that must be executed in the security context of an administrative account--or, worse, as a service running as the Local System account-when, with some thought, the product designers could have not required such privileged accounts. The reason for running with least privilege is quite simple. If a security vulnerability is found in the code and an attacker can inject code into your process, make the code perform sensitive tasks, or run a Trojan horse or virus, the malicious code will run with the same privileges as the compromised process. If the process is running as an administrator, the malicious code runs as an administrator. This is why we recommend people do not run as a member of the local administrators group on their computers, just in case a virus or some other malicious code executes.
>
> Go on, admit it: you're logged on to your computer as a member of the local administrators group, aren't you" I'm not. I haven't been for over three years, and everything works fine. I write code, I debug code, I send e-mail, I sync with my Pocket PC, I create documentation for an intranet site, and do myriad other things. To do all this, you don't need admin rights, so why run as an admin? (I will

admit that when I build a new computer I add myself to the admin group, install all the applications I need, and then promptly remove myself.)

When you create your application, write down what resources it must access and what special tasks it must perform. Examples of resources include files and registry data; examples of special tasks include the ability to log user accounts on to the system, debug processes, or backup data. Often you'll find you do not require many special privileges or capabilities to get any tasks done. Once you have a list of all your resources, determine what might need to be done with those resources. For example, a user might need to read and write to the resources but not create or delete them. Armed with this information, you can determine whether the user needs to run as an administrator to use your application. The chances are good that she does not.

A common use of least privilege again involves banks. The most valued part of a bank is the vault, but the tellers do not generally have access to the vault. That way an attacker could threaten a teller to access the vault, but the teller simply won't know how to do it.

For a humorous look at the principle of least privilege, refer to "If we don't run as admin, stuff breaks" in Appendix B [in *Writing Secure Code*], "Ridiculous Excuses We've Heard." Also, see Chapter 7 [in *Writing Secure Code*] for a full account of how you can often get around requiring dangerous privileges.

Tip: If your application fails to run unless the user (or service process identity) is an administrator or the system account, determine why. Chances are good that elevated privileges are unnecessary.

According to NIST [NIST 01] in Section 3.3, "IT Security Principles," from page 16:

Implement least privilege.

The concept of limiting access, or "least privilege," is simply to provide no more authorizations than necessary to perform required functions. This is perhaps most often applied in the administration of the system. Its goal is to reduce risk by limiting the number of people with access to critical system security controls; i.e., controlling who is allowed to enable or disable system security features or change the privileges of users or programs. Best practice suggests it is better to have several administrators with limited access to security resources rather than one person with "super user" permissions.

Consideration should be given to implementing role-based access controls for various aspects of system use, not only administration. The system security policy can identify and define the various roles of users or processes. Each role is assigned those permissions needed to perform its functions. Each permission specifies a permitted access to a particular resource (such as "read" and "write" access to a specified file or directory, "connect" access to a given host and port, etc.). Unless a permission is granted explicitly, the user or process should not be able to access the protected resource.

According to Schneier [Schneier 00] in "Security Processes":

Limit Privilege.

Don't give any user more privileges than he absolutely needs to do his job. Just as you wouldn't give a random employee the keys to the CEO's office, don't give him a password to the CEO's files.

What Goes Wrong

According to McGraw and Viega [McGraw 03]:[14]

Little problems can become big problems when they happen in privileged sections of code (think SUID code or code that must be run as Administrator to work). Sometimes they're introduced by installation or configuration—something that's impossible for a developer to control. For example, users commonly install a Web server and run it in a real user process space, without creating a

14. All rights reserved. It is reprinted with permission from CMP Media LLC.

nonprivileged "nobody" as the target. Also consider that Solaris SUID binaries can be run without an s-bit set, introducing unacceptable security risk.

Even if you do carefully dole out privilege, relinquishing the privilege isn't always a trivial task. Nevertheless, do it whenever you can.

References

[Bishop 03]	Bishop, Matt. *Computer Security: Art and Science*. Boston, MA: Addison-Wesley, 2003.
[Howard 02]	Howard, Michael & LeBlanc, David. *Writing Secure Code*, *2nd ed*. Redmond, WA: Microsoft Press, 2002.
[McGraw 03]	McGraw, Gary & Viega, John. "Keep It Simple." *Software Development.* CMP Media LLC, May, 2003.
[NIST 01]	NIST. *Engineering Principles for Information Technology Security*. Special Publication 800-27. US Department of Commerce, National Institute of Standards and Technology, 2001.
[Saltzer 75]	Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems," 1278-1308. *Proceedings of the IEEE 63,* 9 (September 1975).
[Schneier 00]	Schneier, Bruce. "The Process of Security[16]." *Information Security Magazine,* April, 2000.
[Viega 02]	Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.

Cigital, Inc. Copyright

1. mailto:copyright@cigital.com

Separation of Privilege

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-12-06 L4 / D/P, L[5]

A system should ensure that multiple conditions are met before granting permissions to an object. Checking access on only one condition may not be adequate for strong security. If an attacker is able to obtain one privilege but not a second, he or she may not be able to launch a successful attack. If a software system largely consists of one component, the idea of having multiple checks to access different components cannot be implemented. Compartmentalizing software into separate components that require multiple checks for access can inhibit an attack or potentially prevent an attacker from taking over an entire system.

Detailed Description Excerpts

According to Saltzer and Schroeder [Saltzer 75] in "Basic Principles of Information Protection" on page 9:

> Separation of privilege: Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key. The relevance of this observation to computer systems was pointed out by R. Needham in 1973. The reason is that, once the mechanism is locked, the two keys can be physically separated and distinct programs, organizations, or individuals made responsible for them. From then on, no single accident, deception, or breach of trust is sufficient to compromise the protected information. This principle is often used in bank safe-deposit boxes. It is also at work in the defense system that fires a nuclear weapon only if two different people both give the correct command. In a computer system, separated keys apply to any situation in which two or more conditions must be met before access should be permitted. For example, systems providing user-extendible protected data types usually depend on separation of privilege for their implementation.

According to Bishop [Bishop 03] in Chapter 13, "Design Principles," in the "Principle of Separation of Privilege" section on pages 347-348:[9]

> This principle is restrictive because it limits access to system entities.
>
> Definition 13-6, The principle of separation of privilege states that a system should not grant permission based upon a single condition.
>
> This principle is equivalent to the separation of duty principle discussed in Section 6.1 [of *Computer Security*]. Company checks for over $75,000 must be signed by two officers of the company. If either does not sign, the check is not valid. The two conditions are the signatures of both officers.
>
> Similarly, systems and programs granting access to resources should do so when more than one condition is met. This provides a fine grained control over the resource, and additional assurance that the access is authorized.
>
> **Example 1.** On Berkeley-based versions of the UNIX operating system, users are not allowed to change from their account to the root account unless two conditions are met. The first is that the user knows the root password. The second is that the user is in the wheel group (the group with GID 0). Meeting either condition is not sufficient to acquire root access. Meeting both conditions is required.

Separation of privilege is defined differently by Howard and LeBlanc [Howard 02]. We include their definition to show the importance of having multiple processes working together with different levels of

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)
9. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.

privileges. This excerpt is from Chapter 3, "Security Principles to Live By," in the "Separation of Privilege" section on pages 61-62:[10]

> An issue related to using least privilege is support for separation of privilege. This means removing high privilege operations to another process and running that process with the higher privileges required to perform its tasks. Day-to-day interfaces are executed in a lower privileged process.
>
> In June 2002, a severe exploit in OpenSSH v2.3.1 and v3.3, which ships with versions of Apple Mac OS X, FreeBSD and OpenBSD, was mitigated in v3.3 because it supports separation of privilege by default. The code that contained the vulnerability ran with lower capabilities because the UsePrlvilegeSeparation option was set in sshd_config. You can read about the issue at http://www.openssh.com/txt/preauth.adv.
>
> Another example or privilege separation is Microsoft Internet Information Services (lIS) 6, which ships in Windows .NET Server. Unlike lIS 5, it does not execute user code in elevated privileges by default. All user mode HTTP requests are handled by external worker processes (named w3wp.exe) that run under the Network Service account, not under the more privileged Local System account. However, the administration and process management process, inetinfo.exe, which has no direct interface to HTTP requests, runs as Local System.
>
> The Apache Web Server is another example. When it starts up, it starts the main Web server process, httpd, as root and then spawns new httpd processes that run as the low privilege nobody account to handle the Web requests.

References

[Bishop 03]	Bishop, Matt. *Computer Security: Art and Science*. Boston, MA: Addison-Wesley, 2003.
[Howard 02]	Howard, Michael & LeBlanc, David. *Writing Secure Code*, *Second Edition*. Redmond, WA: Microsoft Press, 2002.
[McGraw 03c]	McGraw, Gary & Viega, John. "Divide and Conquer." *Software Development.* CMP Media LLC, April, 2003.
[NIST 01]	NIST. *Engineering Principles for Information Technology Security*. Special Publication 800-27. US Department of Commerce, National Institute of Standards and Technology, 2001.
[Saltzer 75]	Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems," 1278-1308. *Proceedings of the IEEE 63,* 9 (September 1975).
[Viega 02]	Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.

Cigital, Inc. Copyright

10. From *Writing Secure Code*, *Second Edition* (0-7356-1722-8) by Microsoft Press. All rights reserved.
1. mailto:copyright@cigital.com

The Build Security In (BSI) portal is sponsored by the U.S. Department of Homeland Security (DHS), National Cyber Security Division. The Software Engineering Institute (SEI) develops and operates BSI. DHS funding supports the publishing of all site content.

Economy of Mechanism

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-09-13 L4 / D/P, L[5]

One factor in evaluating a system's security is its complexity. If the design, implementation, or security mechanisms are highly complex, then the likelihood of security vulnerabilities increases. Subtle problems in complex systems may be difficult to find, especially in copious amounts of code. For instance, analyzing the source code that is responsible for the normal execution of a functionality can be a difficult task, but checking for alternate behaviors in the remaining code that can achieve the same functionality can be even more difficult. One strategy for simplifying code is the use of choke points, where shared functionality reduces the amount of source code required for an operation. Simplifying design or code is not always easy, but developers should strive for implementing simpler systems when possible.

Detailed Description Excerpts

According to Saltzer and Schroeder [Saltzer 75] in "Basic Principles of Information Protection" from page 8:

> Economy of mechanism: Keep the design as simple and small as possible. This well-known principle applies to any aspect of a system, but it deserves emphasis for protection mechanisms for this reason: design and implementation errors that result in unwanted access paths will not be noticed during normal use (since normal use usually does not include attempts to exercise improper access paths). As a result, techniques such as line-by-line inspection of software and physical examination of hardware that implements protection mechanisms are necessary. For such techniques to be successful, a small and simple design is essential.

According to Bishop [Bishop 03] in Chapter 13, "Design Principles," in "Principle of Economy of Mechanism" from pages 344-345:

> This principle simplifies the design and implementation of security mechanisms.
>
> Definition 13-3. The principle of economy of mechanism states that security mechanisms should be as simple as possible.
>
> If a design and implementation are simple, fewer possibilities exist for errors. The checking and testing process is less complex, because fewer components and cases need to be tested. Complex mechanisms often make assumptions about the system and environment in which they run. If these assumptions are incorrect, security problems may result.
>
> Interfaces to other modules are particularly suspect, because modules often make implicit assumptions about input or output parameters or the current system state; should any of these assumptions be wrong, the module's actions may produce unexpected, and erroneous, results. Interaction with external entities, such as other programs, systems, or humans, amplified this problem.
>
> **Example 1**
>
> The ident protocol sends the user name associated with a process that has a TCP connection to a remote host. A mechanism on host A that allows access based on the results of an ident protocol result makes the assumption that the originating host is trustworthy. If host B decides to attack host A, it can connect and then send any identity it chooses in response to the ident request. This is an example of a mechanism making an incorrect assumption about the environment (specifically that host B can be trusted).

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)

Example 2

The finger protocol transmits information about a user or system. Many client implementations assume that the server's response is well-formed. However, if an attacker were to create a server that generated an infinite stream of characters, and a finger client were to connect to it, the client would print all the characters. As a result, log files and disks could be filled up, resulting in a denial of service attack on the querying host. this is an example of incorrect assumptions about the input to the client.

According to Viega and McGraw [Viega 02] in Chapter 5, "Guiding Principles for Software Security," in "Principle 6: Keep It Simple" from pages 104-107:

> The KISS mantra is pervasive: "Keep It Simple, Stupid!" This motto applies just as well to security as it does everywhere else. Complexity increases the risk of problems. Avoid complexity and avoid problems.
>
> The most obvious implication is that software design and implementation should be as straightforward as possible. Complex design is never easy to understand, and is therefore more likely to include subtle problems that will be missed during analysis. Complex code tends to be harder to maintain as well. And most importantly, complex software tends to be far more buggy. We don't think this comes as any big surprise to anyone.
>
> Consider reusing components whenever possible, as long as the components to be reused are believed to be of good quality. The more successful use that a particular component has seen, the more intent you should be on not having to rewrite it. This consideration particularly holds true for cryptographic libraries. Why would anyone want to re-implement AES or SHA-1, when there are several widely used libraries available? Well-used libraries are much more likely to be robust than something put together in-house, since people are more likely to have noticed implementation problems. Experience builds assurance, especially when those experiences are positive. Of course, there is always the possibility of problems even in widely used components, but it's reasonable to suspect that there's less risk involved in the known quantity, all other things [being] equal (although, do refer back to Chapter 4 [in *Building Secure Software*] for several caveats).
>
> It also stands to reason that adding bells and whistles tends to violate the simplicity principle. True enough. But what if the bells and whistles in question are security features? When we discussed defense in depth, we said that we wanted redundancy. Here, we seem to be arguing the opposite. We previously said, "don't put all your eggs in one basket." Now we're saying, "be wary of having multiple baskets." Both notions make sense, even though they're obviously at odds with each other.
>
> The key to unraveling this paradox is to strike a balance that is right for each particular project. When adding redundant features, the idea is to improve the apparent security of the system. Once enough redundancy has been added to address whatever security level is desired, then extra redundancy is not necessary. In practice, a second layer of defense is usually a good idea, but a third layer should be carefully considered.
>
> Despite its face value obviousness, the simplicity principle has its subtleties. Building as simple a system as possible while still meeting security requirements is not always easy. An online trading system without encryption is certainly simpler than an otherwise equivalent one that includes crypto, but there's no way that it's more secure.
>
> Simplicity can often be improved by funneling all security-critical operations through a small number of choke points in a system. The idea behind a choke point is to create a small, easily controlled interface through which control must pass. This is one way to avoid spreading security code throughout a system. In addition, it is far easier to monitor user behavior and input if all users are forced into a few small channels. That's the idea behind having only a few entrances at sports stadiums; if there were too many entrances, collecting tickets would be harder and more staff would be required to do the same quality job.

One important thing about choke points is that there should be no secret ways around them. Harking back to our example, if a stadium has an unsecured chain link fence, you can be sure that people without tickets will climb it. Providing "hidden" administrative functionality or "raw" interfaces to your functionality that are available to savvy attackers can easily backfire. There have been plenty of examples where a hidden administrative backdoor could be used by a knowledgeable intruder, such as a backdoor in the Dansie shopping cart, or a backdoor in Microsoft FrontPage, both discovered in the same month (April, 2000). The FrontPage backdoor became somewhat famous due to a hidden encryption key that read, "Netscape engineers are weenies!"

Another not-so-obvious but important aspect of simplicity is usability. Anyone who needs to use a system should be able to get the best security it has to offer easily, and should not be able to introduce insecurities without thinking carefully about it. Even then, they should have to bend over backwards. Usability applies both to the people who use a program, and to the people who have to maintain its code base, or program against its API.

Many people seem to think they've got an intuitive grasp on what is easy to use. But usability tests tend to prove them wrong. That might be okay for generic functionality, because a given product might be cool enough that ease of use isn't a real concern. When it comes to security, though, usability becomes more important than ever.

Strangely enough, there's an entire science of usability. All software designers should read two books in this field, *The Design of Everyday Things*[9] and *Usability Engineering.*[10] This space is too small to give adequate coverage to the topic. However, we can give you some tips, as they apply directly to security:

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

   Consider enterprise application servers that have encryption turned off by default. Such functionality is usually turned on with a menu option on an administrative tool somewhere. But even if a system administrator stops to think about security, it's likely that person will think, "they certainly have encryption on by default."
2. Talk to users to figure out what their security requirements are. As Jakob Nielson likes to say, corporate vice presidents are NOT users. You shouldn't assume that you know what people will need; go directly to the source. Try to provide users with more security than they think they need.
3. Realize that users aren't always right. Most users aren't well informed about security. They may not understand many security issues, so try to anticipate their needs. Don't give them security dialogs that they can ignore. Err on the side of security. If your assessment of their needs provides more security than theirs, use yours (actually, try to provide more security than you think they need in either case).

   As an example, think about a system such as a web portal, where one service you provide is stock quotes. Your users might never think there's a good reason to secure that kind of stuff at all. After all, stock quotes are for public consumption. But there is a good reason—an attacker could tamper with the quotes users get. Users might decide to buy or sell something based on bogus information, and lose their shirts. Sure, you don't have to encrypt the data; you can use a MAC (Message Authentication Code; see Appendix A [in *Building Secure Software*]). 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 a dialog box that says "WARNING!" in big, bright red letters. To the user, dialog boxes

9. Norman, Donald A. *The Design of Everyday Things*. New York, NY: Basic Books, 2002.
10. Nielsen, Jakob. *Usability Engineering*. San Francisco, CA: Morgan Kaufmann, 1994.

are an annoyance if they keep the user from what he or she wants to do. For example, a lot of mobile code systems (such as web based ActiveX controls) will pop up a dialog box, telling you who signed the code, and asking you if you really want to trust that signer. Do you think anyone actually reads that stuff? Nope. Users just want to get to see the program advertised run, and will take the path of least resistance, without considering the consequences. In the real world, the dialog box is just a big annoyance. As Ed Felten says, "Given the choice between dancing pigs and security, users will pick dancing pigs every time."

Keeping it simple is important in many domains, including security.

We include three relevant principles from NIST [NIST 01] in Section 3.3, "IT Security Principles," from page 16:

Do not implement unnecessary security mechanisms. Every security mechanism should support a security service or set of services, and every security service should support one or more security goals. Extra measures should not be implemented if they do not support a recognized service or security goal. Such mechanisms could add unneeded complexity to the system and are potential sources of additional vulnerabilities. An example is file encryption supporting the access control service that in turn supports the goals of confidentiality and integrity by preventing unauthorized file access. If file encryption is a necessary part of accomplishing the goals, then the mechanism is appropriate. However, if these security goals are adequately supported without inclusion of file encryption, then that mechanism would be an unneeded system complexity.

From page 10:

Strive for Simplicity. The more complex the mechanism, the more likely it may possess exploitable flaws. Simple mechanisms tend to have fewer exploitable flaws and require less maintenance. Further, because configuration management issues are simplified, updating or replacing a simple mechanism becomes a less intensive process.

From page 17:

Strive for operational ease of use. The more difficult it is to maintain and operate a security control, the less effective that control is likely to be. Therefore, security controls should be designed with ease-of-use as an important consideration. The experience and expertise of administrators and users should be appropriate and proportional to the operation of the security control. An organization must invest the resources necessary to ensure system administrators and users are properly trained. Moreover, administrator and user training costs along with the life-cycle operational costs should be considered when determining the cost-effectiveness of the security control.

According to Schneier [Schneier 00] in the section "Security Processes":

Embrace Simplicity. Keep things as simple as absolutely possible. Security is a chain; the weakest link breaks it. Simplicity means fewer links.

According to McGraw and Viega [McGraw 03]:[11]

Security risks can remain hidden due to a system's complexity. By their very nature, complex systems introduce multiple risks—and almost all systems that involve software are complex. One risk is that malicious functionality can be added to a system (either during creation or afterward) that extends it past its intended design. As an unfortunate side effect, inherent complexity lets malicious and flawed subsystems remain invisible to unsuspecting users until it's too late. This is one of the root causes of the malicious code problem. Another risk, more relevant to our purposes, is that a system's complexity makes it hard to understand, hard to analyze and hard to secure. Security is difficult to get right even in simple systems; complex systems serve only to make security more difficult. Security risks can remain hidden in the jungle of complexity, not coming to light until these areas have been exploited.

11. All rights reserved. It is reprinted with permission from CMP Media LLC.

A desktop system running Windows/XP and associated applications depends upon the proper functioning of the kernel as well as the applications to ensure that vulnerabilities can't compromise the system. However, XP itself consists of at least 40 million lines of code, and end-user applications are becoming equally, if not more, complex. When systems become this large, bugs can't be avoided.

The complexity problem is exacerbated by the use of unsafe programming languages (for example, C or C++) that don't protect against simple kinds of attacks, such as buffer overflows. In theory, we can analyze and prove that a small program is free of problems, but this task is impossible for even the simplest desktop systems today, much less the enterprise-wide systems employed by businesses or governments.

References

[Bishop 03] Bishop, Matt. *Computer Security: Art and Science*. Boston, MA: Addison-Wesley, 2003.

[McGraw 03] McGraw, Gary & Viega, John. "Keep It Simple." *Software Development.* CMP Media LLC, May, 2003.

[NIST 01] NIST. *Engineering Principles for Information Technology Security*. Special Publication 800-27. US Department of Commerce, National Institute of Standards and Technology, 2001.

[Saltzer 75] Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems," 1278-1308. *Proceedings of the IEEE 63,* 9 (September 1975).

[Schneier 00] Schneier, Bruce. "The Process of Security[13]." *Information Security Magazine,* April, 2000.

[Viega 02] Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.

Cigital, Inc. Copyright

1. mailto:copyright@cigital.com

Least Common Mechanism

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-09-14 L4 / D/P, L[5]

Avoid having multiple subjects sharing mechanisms to grant access to a resource. For example, serving an application on the Internet allows both attackers and users to gain access to the application. Sensitive information can potentially be shared between the subjects via the mechanism. A different mechanism (or instantiation of a mechanism) for each subject or class of subjects can provide flexibility of access control among various users and prevent potential security violations that would otherwise occur if only one mechanism was implemented.

Detailed Description Excerpts

According to Saltzer and Schroeder [Saltzer 75] in "Basic Principles of Information Protection" from pages 9-10:

> Least common mechanism: Minimize the amount of mechanism common to more than one user and depended on by all users.[9] Every shared mechanism (especially one involving shared variables) represents a potential information path between users and must be designed with great care to be sure it does not unintentionally compromise security. Further, any mechanism serving all users must be certified to the satisfaction of every user, a job presumably harder than satisfying only one or a few users. For example, given the choice of implementing a new function as a supervisor procedure shared by all users or as a library procedure that can be handled as though it were the user's own, choose the latter course. Then, if one or a few users are not satisfied with the level of certification of the function, they can provide a substitute or not use it at all. Either way, they can avoid being harmed by a mistake in it.

According to Bishop [Bishop 03] in Chapter 13, "Design Principles," in 13.2.7, "Principle of Least Common Mechanism," from page 348:[10]

> This principle is restrictive because it limits sharing.
>
> Definition 13-7. The principle of least common mechanism states that mechanisms used to access resources should not be shared.
>
> Sharing resources provides a channel along which information can be transmitted, and so such sharing should be minimized. In practice, if the operating system provides support for virtual machines, the operating system will enforce this privilege automatically. Otherwise, it will provide some support (such as a virtual memory space) but not complete support (because the file system will appear as shared among several processes).
>
> **Example 1**
>
> A web site provides electronic commerce services for a major company. Attackers want to deprive the company of the revenue they obtain from that web site. They flood the site with messages, and tie up the electronic commerce services. Legitimate customers are unable to access the web site and, as a result, take their business elsewhere.

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)
9. G. Popek, "A principle of kernel design," *in 1974 NCC, AFIPS Cong. Proc.*, vol. 43, pp.977-978. (I-A3).
10. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.

Here, the sharing of the Internet with the attackers' sites caused the attack to succeed. The appropriate countermeasure would be to restrict the attackers? access to the segment of the Internet connected to the web site. Techniques to do this include proxy servers such as the Purdue SYN intermediary[11] or traffic throttling. The former targets suspect connections; the latter reduces load on the relevant segment of the network indiscriminately.

References

[Bishop 03] Bishop, Matt. *Computer Security: Art and Science*. Boston, MA: Addison-Wesley, 2003.

[Saltzer 75] Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems," 1278-1308. *Proceedings of the IEEE 63,* 9. IEEE, September 1975.

Cigital, Inc. Copyright

11. C. Schuba, I. Krsul, M. Kuhn, E. Spafford, A. Sundaram, and D. Zamboni, "Analysis of a Denial of Service Attack on TCP," *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 208-223 (May 1997).
1. mailto:copyright@cigital.com

Reluctance to Trust

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-09-15 L4 / D/P, L[5]

Developers should assume that the environment in which their system resides is insecure. Trust, whether it is in external systems, code, people, etc., should always be closely held and never loosely given. When building an application, software engineers should anticipate malformed input from unknown users. Even if users are known, they are susceptible to social engineering attacks, making them potential threats to a system. Also, no system is one hundred percent secure, so the interface between two systems should be secured. Minimizing the trust in other systems can increase the security of your application.

Detailed Description Excerpts

According to Viega and McGraw [Viega 02] in Chapter 5, "Guiding Principles for Software Security," in "Principle 9: Be Reluctant to Trust" from pages 111-112:[9]

> People commonly hide secrets in client code, assuming those secrets will be safe. The problem with putting secrets in client code is that talented end users will be able to abuse the client and steal all its secrets. Instead of making assumptions that need to hold true, you should be reluctant to extend trust. Servers should be designed not to trust clients, and vice versa, since both clients and servers get hacked. A reluctance to trust can help with compartmentalization.
>
> For example, while shrink-wrapped software can certainly help keep designs and implementations simple, how can any off-the-shelf component be trusted to be secure? Were the developers security experts? Even if they were well versed in software security, are they also infallible? There are hundreds of products from security vendors with gaping security holes. Ironically, many developers, architects and managers in the security tool business don't actually know very much about writing secure code themselves. Many security products introduce more risk than they address.
>
> Trust is often extended far too easily in the area of customer support. Social engineering attacks are thus easy to launch against unsuspecting customer support agents, who have a proclivity to trust since it makes their jobs easier.
>
> "Following the herd" has similar problems. Just because a particular security feature is an emerging standard doesn't mean it actually makes any sense. And even if competitors are not following good security practices, you should still consider good security practices yourself. For example, we often hear people deny a need to encrypt sensitive data because their competitors aren't encrypting their data. This argument will hold up only as long as customers are not hacked. Once they are, they will look to blame someone for not being duly diligent about security.
>
> Skepticism is always good, especially when it comes to security vendors. Security vendors all too often spread suspect or downright false data to sell their products. Most snake oil peddlers work by spreading FUD—Fear, Uncertainty and Doubt. Many common warning signs can help identify security quacks. One of our favorites is the advertising of "million bit keys" for a secret-key encryption algorithm. Mathematics tells us that 256 bits will likely be a big enough symmetric key to protect messages through the lifetime of the universe, assuming the algorithm using the key is of high quality. People advertising more know too little about the theory of cryptography to sell worthwhile security products. Before making a security buy decision, make sure to do lots of research. One good

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)
9. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.

> place to start is the "Snake Oil" FAQ, available at http://www.interhack.net/people/cmcurtin/snake-oil-faq.html.
>
> Sometimes it is prudent not to trust even yourself. It is all too easy to be short sighted when it comes to your own ideas and your own code. While everyone wants to be perfect, it is often wise to admit that nobody is and periodically get some objective, high-quality outside eyes to review what you're doing.
>
> One final point to remember is that trust is transitive. Once you dole out some trust, you often implicitly extend it to anyone the trusted entity may trust. For this reason, trusted programs should not invoke untrusted programs, ever. It is also good to be careful when determining whether a program should be trusted or not; see Chapter 12 [in *Building Secure Software*] for a complete discussion.
>
> When you spread around trust, be careful.

According to Howard and LeBlanc [Howard 02] in Chapter 3, "Security Principles to Live By," in "Assume External Systems Are Insecure" from pages 63-64:

> Assuming external systems are insecure is related to defense in depth—the assumption is actually one of your defenses. Consider any data you receive from a system you do not have complete control over to be insecure and a source of attack. This is especially important when accepting input from users. Until you can prove otherwise, all external stimuli have the potential to be an attack.
>
> External servers can also be a potential point of attack. Clients can be redirected in a number of ways to the wrong server. As is covered in more depth in Chapter 15 [of *Writing Secure Code*], "Socket security," the DNS infrastructure we rely on to find the correct server is not very robust. When writing client-side code, do not make the assumption that you're only dealing with a well-behaved server.
>
> Don't assume that your application will always communicate with an application that limits the commands a user can execute from the user interface or Web-based client portion of your application. Many server attacks take advantage of the ease of sending malicious data to the server by circumventing the client altogether. The same issue exists in the opposite direction, clients compromised by rogue servers.

We include two relevant principles from NIST [NIST 01] in Section 3.3, "IT Security Principles," from page 11:

> Minimize the system elements to be trusted.
>
> Security measures include people, operations, and technology. Where technology is used, hardware, firmware, and software should be designed and implemented so that a minimum number of system elements need to be trusted in order to maintain protection. Further, to ensure cost-effective and timely certification of system security features, it is important to minimize the amount of software and hardware expected to provide the most secure functions for the system.

From page 8:

> Assume that external systems are insecure.
>
> The term information domain arises from the practice of partitioning information resources according to access control, need, and levels of protection required. Organizations implement specific measures to enforce this partitioning and to provide for the deliberate flow of authorized information between information domains. An external domain is one that is not under your control.
>
> In general, external systems should be considered insecure. Until an external domain has been deemed "trusted," system engineers, architects, and IT specialists should presume the security measures of an external system are different than those of a trusted internal system and design the system security features accordingly.

According to Bishop [Bishop 03] in Chapter 18, "Introduction to Assurance," in "Assurance and Trust" from pages 477-478:

In previous chapters we have used the terms *trusted system* and *secure system* without defining them precisely. When looked on as an absolute, creating a secure system is an ultimate, albeit unachievable, goal. As soon as we have figured out how to address one type of attack on a system, other types of attacks occur. In reality, we cannot yet build systems that are guaranteed to be secure or to remain secure over time. However, vendors frequently use the term "secure" in product names and product literature to refer to products and systems that have "some" security included in their design and implementation. The amount of security provided can vary from a few mechanisms to specific well-defined security requirements and well-implemented security mechanisms to those requirements. However, providing security requirements and functionality may not be sufficient to engender trust in the system.

Intuitively, trust is a belief or desire that a computer entity will do what it should to protect resources and be safe from attack. However, in the realm of computer security, trust has a very specific meaning. We will define trust in terms of a related concept.

Definition 18-1. An entity is trustworthy if there is sufficient credible evidence leading one to believe that the system will meet a set of given requirements. Trust is a measure of trustworthiness, relying on the evidence provided.

These definitions emphasize that calling something "trusted" or "trustworthy" does not make it so. Trust and trustworthiness in computer systems must be backed by concrete evidence that the system meets its requirements, and any literature using these terms needs to be read with this qualification in mind. To determine trustworthiness, we focus on methodologies and metrics that allow us to measure the degree of confidence that we can place in the entity under consideration. A different term captures this notion.

Definition 18-2. Security assurance, or simply assurance, is confidence that an entity meets its security requirements, based on specific evidence provided by the application of assurance techniques.

"What Goes Wrong"

According to MGraw and Viega [McGraw 03]:[13]

> Developers like to know what's going on in their programs, especially when a program does something unforeseen, like cause an error. For this reason, some coders tend to put diagnostic information about errors directly in error messages displayed to the user. Attackers can and will use this information to exploit your code. In fact, clever attackers combine trusted input risks with blabbermouth error-reporting to exploit software. Put diagnostic information that could be used in an exploit somewhere safe (like an error log file), instead.

References

[Bishop 03]	Bishop, Matt. *Computer Security: Art and Science*. Boston, MA: Addison-Wesley, 2003.
[Howard 02]	Howard, Michael & LeBlanc, David. *Writing Secure Code, 2nd ed*. Redmond, WA: Microsoft Press, 2002.
[McGraw 03]	McGraw, Gary & Viega, John. "Keep It Simple." *Software Development.* CMP Media LLC, May, 2003.
[NIST 01]	NIST. *Engineering Principles for Information Technology Security*. Special Publication 800-27. US Department of Commerce, National Institute of Standards and Technology, 2001.
[Viega 02]	Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.

13. All rights reserved. It is reprinted with permission from CMP Media LLC.

Cigital, Inc. Copyright

The Build Security In (BSI) portal is sponsored by the U.S. Department of Homeland Security (DHS), National Cyber Security Division. The Software Engineering Institute (SEI) develops and operates BSI. DHS funding supports the publishing of all site content.

1. mailto:copyright@cigital.com

Never Assuming That Your Secrets Are Safe

Sean Barnum, Cigital, Inc. [vita[4]]
Michael Gegick, Cigital, Inc. [vita[5]]

2005-09-14 L4 / D/P, L[6]

Relying on an obscure design or implementation does not guarantee that a system is secured. You should always assume that an attacker can obtain enough information about your system to launch an attack. Tools such as decompilers and disassemblers allow attackers to obtain sensitive information that may be stored in binary files. Also, inside attacks, which may be accidental or malicious, can lead to security exploits. Using real protection mechanisms to secure sensitive information should be the ultimate means of protecting your secrets.

Detailed Description Excerpts

According to Howard and LeBlanc [Howard 02] in Chapter 3, "Security Principles to Live By," in "Never Depend on Security Through Obscurity Alone" from pages 66-67:

> Always assume that an attacker knows everything that you know -- assume the attacker has access to all source code and all designs. Even if this is not true, it is trivially easy for an attacker to determine obscured information. Other parts of this book show many examples of how such information can be found. Obscurity is a useful defense, so long as it is not your only defense. In other words, it's quite valid to use obscurity as a small part of an overall defense in depth strategy.

We include two relevant principles from Viega and McGraw [Viega 02]. In Chapter 5, "Guiding Principles for Software Security," in "Principle 8: Remember that Hiding Secrets Is Hard" from pages 109-111:[10]

> Security is often about keeping secrets. Users don't want their personal data leaked. Keys must be kept secret to avoid eavesdropping and tampering. Top-secret algorithms need to be protected from competitors. These kinds of requirements are almost always high on the list, but turn out to be far more difficult to meet than the average user may suspect.
>
> Many people make an implicit assumption that secrets in a binary are likely to stay secret, maybe because it seems very difficult to extract secrets from a binary. It is true that binaries are complex. However, as we discuss in Chapter 4 [in *Building Secure Software*], keeping the "secrets" secret in a binary is incredibly difficult. One problem is that some attackers are surprisingly good at reverse engineering binaries. They can pull a binary apart, and figure out what it does. The transparent binary problem is why copy protection schemes for software tend not to actually work. Skilled youths will circumvent any protection that a company tries to hardcode into their software, and release "cracked" copies. For years, there was an arms race and an associated escalation in techniques of both sides; vendors would try harder to keep people from finding the secrets to "unlock" software, and the software crackers would try harder to break the software. For the most part, the crackers won. Cracks for interesting software like DVD viewers or audio players tend to show up on the same day that the software is officially released, and sometimes sooner.
>
> It may appear that software running server-side on a dedicated network could keep secrets safe, but that's not necessarily the case. Avoid trusting even a dedicated network if possible. Think through a scenario in which some unanticipated flaw lets an intruder steal your software. This actually happened to Id software right before they released the first version of Quake.

4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
5. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)
10.

> Even the most secure networks are often amenable to insider attacks. Several studies show 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, and a friend goes prodding around where she shouldn't. Think about the fact that many companies are not able to protect their firewall-guarded software from a malicious janitor. If someone is really intent on getting to software through illegal means, it can probably be done. When we point out the possibility of an insider attack to clients, we often hear "That won't happen to us; we trust our employees." But relying on this reasoning is dangerous even though 95% of the people we talk to say the same thing. Given that most attacks are perpetrated by insiders, there's a large logical gap here, suggesting that most of the people who believe they can trust their employees must be wrong. Remember that 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 relationship. The moral here is that it pays to be paranoid.
>
> The infamous FBI spy Richard P. Hanssen carried out the ultimate insider attack against U.S. classified networks for over 15 years. Hanssen was assigned to the FBI counterintelligence squad in 1985, around the same time he became a traitor to his country. During some of that time, he was root on a UNIX system. The really disturbing thing is that Hanssen created code (in C and Pascal) that was (is?) used to carry out various communications functions in the FBI. Apparently he wrote code used by agents in the field to cable back to the home office. We sincerely hope that any and all code used in such critical functions is carefully checked before it becomes widely used. If not, the possibility that a Trojan is installed in the FBI comm system is extremely high. Any and all code that was ever touched by Hanssen needs to be checked. In fact, Hanssen sounds like the type of person who might even be able to create hard-to-detect distributed attacks that use covert channels to leak information out.
>
> Software is a powerful tool, both for good and for evil. Since most people treat software as "magic" and never actually look at its inner-workings, the potential for serious misuse and abuse is a very real risk.
>
> Keeping secrets is hard, and is almost always a source of security risk.

In Chapter 5, "Guiding Principles for Software Security," in "Principle 10: Use Your Community Resources" from pages 112-113:

> While it's not a good idea to follow the herd blindly, there is something to be said for strength in numbers. Repeated use without failure promotes trust. Public scrutiny does as well.
>
> For example, in the cryptography field it is 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 large enough number of smart people have spent a lot of time trying to break them, and all fail to make substantial progress.
>
> Many developers find it exciting to write their own cryptographic algorithms, sometimes banking on the fact that if they are weak, security by obscurity will help them. Repeatedly, these sorts of hopes are dashed on the rocks (for two examples, recall 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 why it is not a good idea to expect any algorithm to stay secret for very long. The RC2 and RC4 encryption algorithms, for example, were supposed to be RSA Security trade secrets. However, they were both ultimately reverse engineered and posted anonymously to the Internet.
>
> In any case, cryptographers design their algorithms so that knowledge of the algorithm is unimportant to its security properties. Good cryptographic algorithms work because they rely on keeping a small piece of data secret (the key), not because the algorithm itself is secret. That is, the only thing a user needs to keep private is the key. If a user can do that, and the algorithm is actually good (and the key is long enough), then even an attacker intimately familiar with the algorithm will be unable to break the crypto (given reasonable computational resources).
>
> Similarly, it's far better to trust security libraries that have been widely used, and widely scrutinized. Of course, they might contain bugs that haven't been found. But at least it is possible 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 the components you want to use. As we discussed at length in Chapter 4 [in *Building Secure Software*], one common fallacy is to believe 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 and design that many would like to believe exists. For example, many security bugs in widely used pieces of free software have gone unnoticed for years. In the case of the most popular FTP server around, several security bugs went unnoticed for over a decade!

References

[Howard 02] Howard, Michael & LeBlanc, David. *Writing Secure Code, 2nd ed*. Redmond, WA: Microsoft Press, 2002.

[Viega 02] Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.

Cigital, Inc. Copyright

The Build Security In (BSI) portal is sponsored by the U.S. Department of Homeland Security (DHS), National Cyber Security Division. The Software Engineering Institute (SEI) develops and operates BSI. DHS funding supports the publishing of all site content.

1. mailto:copyright@cigital.com

Complete Mediation

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-09-12 L4 / D/P, L[5]

A software system that requires access checks to an object each time a subject requests access, especially for security-critical objects, decreases the chances of mistakenly giving elevated permissions to that subject. A system that checks the subject's permissions to an object only once can invite attackers to exploit that system. If the access control rights of a subject are decreased after the first time the rights are granted and the system does not check the next access to that object, then a permissions violation can occur. Caching permissions can increase the performance of a system, but at the cost of allowing secured objects to be accessed.

Detailed Description Excerpts

From Saltzer and Schroeder [Saltzer 75], Section: "Basic Principles of Information Protection" on page 9:

> Complete mediation: Every access to every object must be checked for authority. This principle, when systematically applied, is the primary underpinning of the protection system. It forces a system-wide view of access control, which in addition to normal operation includes initialization, recovery, shutdown, and maintenance. It implies that a foolproof method of identifying the source of every request must be devised. It also requires that proposals to gain performance by remembering the result of an authority check be examined skeptically. If a change in authority occurs, such remembered results must be systematically updated.

From Bishop [Bishop 03], Chapter 13: "Design Principles," Section 13.2.4: "Principle of Complete Mediation," pages 345-346:[9]

> This principle restricts the caching of information. This often leads to simpler implementations of mechanisms.
>
> Definition 13-4.The principle of complete mediation requires that all accesses to objects be checked to ensure they are allowed.
>
> Whenever a subject attempts to read an object, the operating system should mediate the action. First, it determines if the subject can read the object. If so, it provides the resources for the read to occur. If the subject tries to read the object again, the system should again check that the subject can still read the object. Most systems would not make the second check. They would cache the results of the first check, and base the second access upon the cached results.
>
> **Example 1**
>
> When a UNIX process tries to read a file, the operating system determines if the process is allowed to read the file. If so, the process receives a file descriptor encoding the allowed access. Whenever the process wants to read the file, it presents the file descriptor to the kernel. The kernel then allows the access. If the owner of the file disallows the process permission to read the file after the file descriptor is issued, the kernel still allows access. This scheme violates the principle of complete mediation, because the second access is not checked. The cached value is used, resulting in the denial of access being ineffective.
>
> **Example 2**

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)
9. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.

The Directory Name Service (DNS) caches information mapping hostnames into IP addresses. If an attacker is able to "poison" the cache by implanting records associating a bogus IP address with a name, the host will route connections to that host incorrectly. Section 14.6.1.2 [of *Computer Security: Art and Science*] discusses this in more detail.

Further Reading

We discuss race conditions and canonicalization as examples where a simple check on an object's permissions or name may not be adequate when performing a sensitive operation.

Race Conditions that Can Circumvent Access Control Checks

Even though multiple access checks may be implemented in their source code, developers should be warned that their security concerns are not necessarily over. Attackers can make use of race conditions that occur between an access check and the subsequent privileged operation. This type of race condition is called time of check, time of use (TOCTOU) and has been exploited in the UNIX operating system. The following code belongs to a program running setuid root [Viega 02]:

```
/* access returns 0 on success */
if (!access(file, W_OK)) {
  f = fopen(file, "wb+");
  write_to_file(f);
} else {
  fprintf(stderr, "Permission denied when trying to open %s. \n", file);
}
```

Users can request that this program write to the files they own. The program first checks to determine whether the real UID has permissions for the request and, if so, returns 0. In the time between when access privileges are checked and when the file is opened, an attacker can replace the file he or she has rights to with a file that is owned by root. An attacker can perform the swap by creating a dummy file (e.g., /etc/passwd) with his or her permission and then creating a symbolic link to it. The attacker can then perform a command such as [Viega 02]

```
$ rm pointer; ln -s /etc/passwd pointer
```

It is then possible that the program will overwrite the system password file [Viega 02]. Therefore, developers should be cautioned to make sure their access checks cannot be subverted with race conditions such as those in the class of TOCTOU.

Careful with Canonicalization

Upon checking a resource for permissions, the name of that resource should correctly identify the object you expect to check. Different names can be used to apply to one object. For example, a file can be named c:\dir\trip.gif, trip.gif, and ..\..\trip.gif. The process of resolving various equivalent forms of a filename into a standard name is called *canonicalization*. The final (or canonical) name in this example could be c:\dir\trip.gif [Howard 02].

An example of where filenames were represented differently to subvert security mechanisms was in the Napster application. Napster implemented filters to restrict selected songs from being shared, in accordance with a federal judge's ruling. Users began to change the filenames of the songs that were ordered not to be shared. For example, one could possibly translate song titles into leet-speak (e.g., Cold Play's "White Shadows"to "VV]-[!73 5|-|4|)0VV2" or Audioslave's "Be Yourself" into "|33 ?/()|_||2531|^}-{"). The filtering mechanisms for those songs did not adequately address the canonicalization of restricted song titles [Howard 02].

References

[Bishop 03]	Bishop, Matt. *Computer Security: Art and Science*. Boston, MA: Addison-Wesley, 2003.
[Howard 02]	Howard, Michael & LeBlanc, David. *Writing Secure Code, 2nd ed*. Redmond, WA: Microsoft Press, 2002.
[Saltzer 75]	Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems." 1278-1308. *Proceedings of the IEEE 63,* 9 (September 1975).
[Viega 02]	Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.

Cigital, Inc. Copyright

The Build Security In (BSI) portal is sponsored by the U.S. Department of Homeland Security (DHS), National Cyber Security Division. The Software Engineering Institute (SEI) develops and operates BSI. DHS funding supports the publishing of all site content.

1. mailto:copyright@cigital.com

Psychological Acceptability

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-09-15 L4 / D/P, L[5]

Accessibility to resources should not be inhibited by security mechanisms. If security mechanisms hinder the usability or accessibility of resources, then users may opt to turn off those mechanisms. Where possible, security mechanisms should be transparent to the users of the system or at most introduce minimal obstruction. Security mechanisms should be user friendly to facilitate their use and understanding in a software application.

Detailed Description Excerpts

According to Saltzer and Schroeder [Saltzer 75] in "Basic Principles of Information Protection" from page 10:

> Psychological acceptability: It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly. Also, to the extent that the user's mental image of his protection goals matches the mechanisms he must use, mistakes will be minimized. If he must translate his image of his protection needs into a radically different specification language, he will make errors.

According to Bishop [Bishop 03] in Chapter 13, "Design Principles," in "Principle of Psychological Acceptability" from pages 348-349:[9]

> This principle recognizes the human element in computer security.
>
> Definition 13-8. The principle of psychological acceptability states that security mechanisms should not make the resource more difficult to access than if the security mechanisms were not present.
>
> Configuring and executing a program should be as easy and as intuitive as possible, and any output should be clear, direct, and useful. If security-related software is too complicated to configure, system administrators may unintentionally set up the software in a non-secure manner. Similarly, security-related user programs must be easy to use and output understandable messages. If a password is rejected, the password changing program should state why it is rejected rather than giving a cryptic error message. If a configuration file has an incorrect parameter, the error message should describe the proper parameter.
>
> On the other hand, security requires that the messages impart no unnecessary information.
>
> In practice, the principle is interpreted to mean that the security mechanism may add some extra burden, but that burden must be both minimal and reasonable.
>
> **Example 1**
>
> The ssh program[10] allows a user to set up a public key mechanism for enciphering communications between systems. The installation and configuration mechanisms for the UNIX version allow one to arrange that the public key be stored locally without any password protection. In this case, one need not supply a password to connect to the remote system, but still obtains the enciphered connection.

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)
9. All rights reserved. It is reprinted with permission from Addison-Wesley Professional.
10. Wulf, W.; Cohen, E.; Corwin, W.; Jones, A.; Levin, R.; Pierson, C.; & Pollack, F. "HYDRA: The Kernel of a Multiprocessor System." *Communications of the ACM 17*, 6 (June 1974): 337-345.

This mechanism satisfies the principle of psychological acceptability.

Example 2

When a user supplies the wrong password during login, the system should reject the attempt with a message stating that the login failed. If it were to say that the password was incorrect, the user would know that the account name was legitimate. If the ?user? were really an unauthorized attacker, she now knows an account for which she can try to guess a password.

Example 3

A mainframe system allows users to place passwords on files. Accessing the files requires that the program supply the password. Although this mechanism violates the principle as stated, it is considered sufficiently minimal to be acceptable. On an interactive system, where the pattern of file accesses is more frequent and more transient, this requirement would be too great a burden to be acceptable.

References

[Bishop 03]	Bishop, Matt. *Computer Security: Art and Science*. Boston, MA: Addison-Wesley, 2003.
[Saltzer 75]	Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems," 1278-1308. *Proceedings of the IEEE 63,* 9 (September 1975).

Cigital, Inc. Copyright

1. mailto:copyright@cigital.com

Promoting Privacy

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-09-14 L4 / D/P, L[5]

Protecting software systems from attackers that may obtain private information is an important part of software security. If an attacker breaks into a software system and steals private information about a vendor's customers, then their customers may lose their confidence in that software system. Attackers may also target sensitive system information that can supply an attacker with the details needed to attack that system. Preventing attackers from accessing private information or obscuring that information can alleviate the risk of information leakage.

Detailed Description Excerpts

According to Viega and McGraw [Viega 02] in Chapter 5, "Guiding Principles for Software Security," in "Principle 7: Promote Privacy" from pages 107-109:[9]

> Many users consider privacy a security concern. Try not to do anything that might compromise the privacy of the user. Be as diligent as possible in protecting any personal information a user does give your program. You've probably heard horror stories about malicious hackers accessing entire customer databases through broken web servers. It's also common to hear about attackers being able to hijack other user's shopping sessions, and thus get at private information. Try really hard to avoid the privacy doghouse. There is often no quicker way to lose customer respect than to abuse user privacy.
>
> One of the things privacy most often trades off against is usability. For example, most systems are better off forgetting about credit card numbers as soon as they are used. That way, even if the web server is compromised, there is not anything interesting stored there long-term. Users hate that solution, though, because it means they have to type in their credit card info every time they want to buy something. A more secure approach would make the convenience of "one-click shopping" impossible.
>
> The only acceptable solution in this case is to store the credit card information, but be really careful about it. We should never show the user any credit card number, even the one that belongs to the user, in case someone manages to get access they shouldn't have. A common solution is to show a partial credit-card number, with most of the digits blacked out. Though not perfect, this compromise is often acceptable.
>
> A better idea might be to ask for the issuing bank, never showing any part of the actual number once it has been captured. The next time the user wants to select a credit card, it can be done by reference to the bank. If a number needs to be changed, because there is a new card, it can be entered directly.
>
> On the server side, the credit card number should be encrypted before being stored in a database. Keep the key for the credit card number on a different machine (this requires decrypting and encrypting on a machine other than the one on which the database lives). That way, if the database gets compromised, then the attacker still needs to find the key, which requires breaking into another machine. This raises the bar.
>
> User privacy isn't the only kind of privacy. Malicious hackers tend to launch attacks based on information easily collected from a hacked system. Services running on a target machine tend to give

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/345-BSI.html (Gegick, Michael)
9.

out lots of information about themselves that can help the attacker figure out how to break in. For example, the telnet service often supplies the operating system name and version:

```
$ telnet testbox
Trying 10.1.1.2...
Connected to testbox (10.1.1.2).
Escape character is '^]'.
```

```
Red Hat Linux release 6.0 (Hedwig)
Kernel 2.2.16 on an i686
login:
```

There is no reason to give out any more information than necessary. First off, a firewall should block unnecessary services, so that an attacker can get no information from them. Second, whether protected by a firewall or not (defense-in-depth, remember?), remove as much publicly provided versioning information from your software as possible. For the example above, the telnet login can be made not to advertise the operating system.

In fact, why not lie about this sort of information? It doesn't hurt to send a potential attacker on a wild goose chase by advertising a Linux box as a Solaris machine. Remote Solaris exploits don't tend to work against a Linux box. The attacker might give up in frustration long before figuring out that the service was set up to lie. (Of course, lies like this make it harder to audit your own configuration.)

Leaving any sort of information around about a system can help potential attackers. Deterring an attacker through misinformation can work. If you're using Rijndael as an encryption algorithm, does it hurt to claim to be using Twofish? Both algorithms are believed to provide similar levels of cryptographic security, so there's probably no harm done.

Attackers collecting information from a software environment can make use of all sorts of subtle information. For example, known idiosyncrasies of different web server software can quickly tell a bad guy what software a company runs, even if the software is modified not to report its version. Usually, it's impossible to close up every such information channel in a system. This type of problem is probably the most difficult security issue to identify and remove from a system. When it comes to privacy, the best strategy is to try to identify the most likely information channels, and use them to your advantage, by sending potential attackers on a wild goose chase.

Promote privacy for your users, for your systems, and for your code.

What Goes Wrong

According to McGraw and Viega [McGraw 03a]:[12]

Finding and breaking passwords can be a trivial task.

Passwords are everywhere, and they're a nuisance! Say you're writing an e-commerce application that makes use of a back-end database that is protected by a DBA password. It might be tempting to hard-code the DBA credentials (username and password) into your code so you don't have to bug your user, but doing this creates a major security flaw: Now anyone who has a copy of the program has a copy of the DBA password.

Consider a security problem discovered in Netscape Communicator near the end of 1999, affecting Netscape mail users who chose to save their POP mail password with the mail client. Obviously, saving a user's mail password means storing it somewhere permanent, but where and how was the information stored? This is the sort of question that potential attackers pose all the time.

Clearly, Netscape's programmers needed to prevent casual users (including attackers) from reading the password directly off the disk, but they also had to provide access to the password for the program that used it in the POP protocol. In an attempt to solve this problem, Netscape's designers tried to

12. All rights reserved. It is reprinted with permission from CMP Media LLC.

encrypt the password before storing it, making it unreadable (in theory, anyway). The programmers chose a "password encryption algorithm" that they believed was good enough, considering that the source code wasn't to be made available. Unfortunately, on Windows machines, the "encrypted" password was stored in the registry, and the "crypto" was really a magic decoder-ring variant (not algorithmically strong). As a result, finding and breaking the password was a trivial task—in fact, attackers wrote a little program that decrypted actual Netscape POP passwords in a split second.

References

[McGraw 03a] McGraw, Gary & Viega, John. "The One-Click Trick." *Software Development.* CMP Media LLC, June, 2003.

[Viega 02] Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.

Cigital, Inc. Copyright

1. mailto:copyright@cigital.com