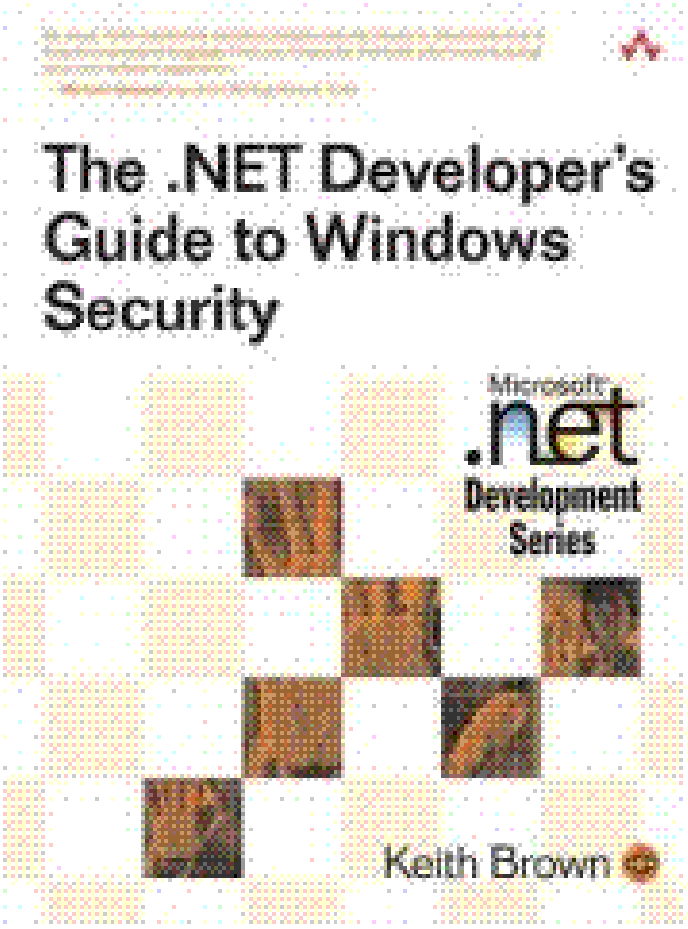
The .NET Developer's Guide to Windows Security

by Keith Brown



Preface

This book was written for the many thousands of people involved in designing and writing software for the Microsoft .NET platform. It is chock-full of tips and insights about user-based security, which I like to term "Windows security" because it's been around in one form or another since Windows NT first shipped. Given the plethora of books that cover the new security features in the .NET Framework, such as code access security and ASP.NET forms authentication, I decided to write a book to help folks with the basics of Windows security, a topic that most other books miss entirely or get subtly or blatantly wrong. This book is in some sense a second edition of my first security book, Programming Windows Security, but I hope that you will find it immensely more approachable and practical. I've tried to distill the Zen of these topics into small tidbits of information— items that link to one another— allowing you to read the book in any order that suits you. I hope that you'll find the format of 75 concise tidbits of information helpful as a reference. The "what is" items focus on explaining concepts, while the "how to" items focus on helping you perform a common task.

Within these pages I cover security features in various versions of Windows based on Windows NT. This includes Windows 2000, Windows XP Professional, and Windows Server 2003, but does not include 16-bit Windows or any of the Win9X flavors (Windows 95/98, Windows ME, Windows XP Home edition). So, when I talk about "Windows" I'm referring to the versions based on Windows NT. Whenever I talk about the file system, I'm assuming that you're using NTFS, not FAT partitions. Whenever I talk about domains, I'm assuming Windows 2000 or greater. If you're still living with a Windows NT 4 domain, you have my sincere condolences!

Many people have expressed surprise that I occasionally talk about Win32 APIs and refer to Win32 header files in a book for .NET programmers. I wish I didn't have to do this, but as anyone who has experience with the .NET Framework knows, the framework class library wraps only a fraction of the functionality of the Windows platform as of this writing. The coverage will get better over time, but to do many things in Windows (including security programming), you often need to call native Win32 APIs. Even as version 2.0 of the framework is being revealed in beta 1, you can see that coverage increasing, but it's still not complete. In any case, I've tried to make it clear in the prose when I'm talking about a Win32 API versus a .NET Framework class, and I've provided lots of sample code and helper classes written in Managed C++ that you can leverage to avoid having to call those APIs yourself.

This book can be found online (in its entirety) in hyperlinked form on the Web at winsecguide.net, where I believe you'll find it to be a great reference when you're connected. I plan to continue filling in more items over time, so subscribe to the RSS feed on the book for news. You can also download samples and tools that I mention in the book from this Web site. Errata will be posted to this site as well, so if you find a problem please let me know.

Good luck in your endeavors!

Keith Brown

Highlands Ranch, CO

http://www.pluralsight.com/keith

Acknowledgements

Thanks to my technical reviewers: John Lambert, Peter Partch, and Bill Moseley. The book wouldn't be the same without your efforts.

I'd like to say a special thank you to Don Box, who jumpstarted my writing and teaching career back in 1997 when he invited me to teach COM for the training company he founded. It was Don who helped me land my column with Microsoft Systems Journal. He encouraged me to work on security back when nobody seemed to care about the topic. I'm still using his Word template when I write articles for MSDN Magazine.

Thanks to all of the people who read the online version of the book before it was published and took the time to e-mail in suggestions. Lots of the tips in the section on running as non-admin came from these folks.

Thanks to Chris Sells for his simple suggestion before I even started writing, "Please give me something practical," he asked.

Thanks to all of my students over the years. Your questions and insights have challenged and strengthened me. Please come up and say hello if you see me at an event. Stay in touch!

Thanks to the folks at Addison-Wesley for their help in getting this book off the ground. Karen Gettman, my editor, didn't let me slip (well, not much at least). Thanks for giving me the leeway I needed to find this rather off-the-wall format for the book. Thanks to the two Elizabeths at AW for all your help getting the book through production, and Connie Leavitt at Bookwrights for managing the copyediting process, even as I submitted entirely new content after beta 1 shipped. Thanks to Curt Johnson and his staff who somehow figure out how to sell all these paperweights I've been writing over the years.

Part 1: The Big Picture

1.1 What is secure code?

One of the major goals of this book is to help clarify how Windows security works so you'll be able to use it effectively in your applications and also in your everyday life. But even if you have a perfect understanding of all the security features of the platform, and make all the right API calls and configure security policy very carefully to keep out attackers, if you don't write your code with security in mind, none of that will matter because you'll still be vulnerable to attack.

Look at the following C# method and count the number of security APIs that it uses.

```
// this code has a really nasty security flaw
void LogUserName(SqlConnection conn, string userName) {
  string sqlText = "insert user_names values('" + userName + "')";
  SqlCommand cmd = new SqlCommand(sqlText, conn);
  cmd.ExecuteNonQuery();
}
```

That's right, it doesn't call any security APIs. However, if we assume the userName parameter has been given to us by someone we don't fully trust (aka a user of our application) then this benign-looking code has a horrible security flaw. If the above function had been written with security in mind, here's how it might have looked instead:

```
// much more secure code
void LogUserName(SqlConnection conn, string userName) {
  string sqlText = "insert user_names values(@n)";
  SqlCommand cmd = new SqlCommand(sqlText, conn);
  SqlParameter p = cmd.Parameters.Add("@n",
      SqlDbType.VarChar, userName.Length);
  p.Value = userName;
  cmd.ExecuteNonQuery();
}
```

Note the difference in the coding style. In the first case, the coder appended untrusted user input directly into a SQL statement. In the second case, the coder hardcoded the SQL statement and encased the user input in a parameter that was sent with the query, carefully keeping any potential attackers in the data channel and out of the control channel (the SQL statement in this case).

The flaw in the first bit of code is that a user with malicious intent can take control of our SQL statement and do pretty much whatever he wants with the database. We've allowed an attacker to slip into a control channel. For example, what if the user were to submit the following string as a user name?

```
SeeYa');drop table user_names--
```

Our SQL statement would now become

```
insert user_names values('SeeYa');drop table user_names--')
```

This is just a batch SQL query with a comment at the end (that's what the -- sequence is for) that inserts a record into the user_names table and then drops that same table from the database! This is a rather extreme example (your database connection should use least privilege so that dropping tables is never allowed anyway; see WhatIsThePrincipleOfLeastPrivilege), but it dramatically emphasizes that the attacker has taken control of your SQL statement and can submit arbitrary SQL to your database. This is really bad![1]

There are many examples where malicious user input can lead to program failure or security breaks. If you're not familiar with things like cross-site scripting, buffer overflow vulnerabilities, and other attacks via malicious user input, please stop reading now and go buy a copy of a book for example, Howard and LeBlanc, 2002 or Viega and McGraw, 2002 that focuses on these sorts of vulnerabilities. Study it, seriously. Perform regular code reviews to keep your sofware free from such bugs. These bugs aren't the focus of this book, but so many developers are unaware of them that I'd be remiss not to mention them here.

It's not enough to know how about security technologies. You need to be able to write secure code yourself.

[1] For more information on exploiting a SQL injection vulnerability, see http://www.issadvisor.com/columns/SqlInjection3/sql-injection-3-exploit-tables_files/frame.htm

1.2: What is a countermeasure?

In his book *Secrets and Lies*, Bruce Schneier talks about countermeasures in three categories: **protection**, **detection**, and **reaction**.

In a military office, classified documents are stored in a safe. The safe provides protection against attack, but so does the system of alarms and guards. Assume that the attacker is an outsider—someone who doesn't work in the office. If he's going to steal the documents inside the safe, he's not only going to have to break into the safe, but he is also going to have to defeat the alarms and guards. The safe—both the lock and the walls—is a protective countermeasure; the guards are reactive countermeasures.

If guards patrol the offices every 15 minutes, the safe only has to withstand attack for a maximum of 15 minutes. If the safe is in an obscure office that is staffed only during the day, it has to withstand 16 hours of attack: from 5 P.M. until 9 A.M. the next day (much longer if the office is closed during holiday weekends). If the safe has an alarm on it, and the guards come running as soon as the safe is jostled, then the safe has to survive attack only for as long as it takes for the guards to respond.

Can you see the synergy of the three types of countermeasure employed in the scenario Bruce describes here? First we have the safe, which is purely a protection countermeasure. The alarms on it provide detection, and the guards provide reaction. Imagine that we didn't have the alarms or guards: The safe would have to be perfect. But as we strengthen the detection and reaction countermeasures, we can rely less on the protection countermeasure. The safe is needed only to buy time for detection and reaction to kick in. Underwriters Laboratories publishes a standard burglary classification for safes[1] that ranges from TL-15, "tool-resistant," to TXTL-60X6, "torch-, explosive, and tool-resistant." But notice the numbers. A TL-15 safe isn't designed to withstand attack forever. It's designed to withstand 15 minutes of sustained attack by someone who knows exactly how the safe is constructed. The TXTL-60X6 rating provides 60 minutes of protection[2]. You're literally buying time.

Think about protection, detection, and reaction in a typical computer system. You might have to think hard to come up with any detection and reaction countermeasures because the focus is almost always on protection. The hardware of the machine provides isolation between processes. This is protection. Cryptography is the basis for even more protection: data integrity protection, authentication, protection from eavesdropping, and so on. Further protection is on the horizon with Microsoft's proposed Next Generation Secure Computing Base (NGSCB).

Intrusion detection systems (IDSs) like Snort (http://www.snort.org) and integrity management systems like Tripwire (http://www.tripwire.com) are examples of detection countermeasures in computer systems, and the latter has some automated reaction built into it, automatically restoring files that have been corrupted. But generally reaction is provided by a human. When the IDS sends an alert to an administrator, someone's got to be on duty to notice and react.

Reaction is an interesting idea, and sometimes we can build it into systems automatically. For example, a domain controller can lock out an account after several failed login attempts, automatically foiling password-guessing attacks (note that this also introduces the potential for a denial-of-service attack). One way to think about reaction is that it allows you to dynamically change the balance between security and usability. The Windows TCP stack is another good example of automatic reaction. It can detect when a SYN-flood attack[3] occurs and react by reducing timeout durations for half-open TCP connections. Thus the system becomes a little bit harder to use (the timeout for acknowledgment is shorter) but is more resistant to attack.

I fear we may have been lulled into designing systems that are based on protection countermeasures alone, and that's not a good idea because we'll never achieve perfect protection and still have systems that are accessible. For example, because we have such great cryptography technology today, people are often lulled into a false sense of security. It often doesn't matter what cryptographic algorithm you happen to be using; as long as it's a reasonably trustworthy algorithm that's been looked at by the cryptographic community, it's probably going to be the strongest link in your security chain. The attacker isn't going to go after the strongest link. He'll look for a weaker point instead.

So, when you design secure systems, try to think of protection countermeasures as a jeweller thinks of a safe. They exist to buy you time. Design detection and reaction into your systems as well. For example, you could instrument your server processes with WMI (Windows Management Instrumentation) (Turstall and Cole 2003) and then use WMI to report security statistics directly to an administrator. You could further build WMI consumers that analyze statistics and automatically react, or provide further alerts to the administrator. This is an area we all need to be working harder to perfect.

[1] http://ulstandardsinfonet.ul.com/scopes/0687.html

[2] The "X6" designation indicates that all six walls of the safe provide the same level of protection. This is a very expensive safe!

[3] A SYN-flood attack is denial of service by repetitively sending the first leg of the TCP handshake (a "SYN" packet) with a spoofed source IP address. The victim thinks someone is trying to open a connection and sends an acknowledgment, then waits for a final acknowledgment from the sender. By flooding the victim with SYN packets from random spoofed IP addresses, the attacker keeps the victim's kernel so busy it can't process legitimate connection requests.

1.3 What is threat modeling?

Security is a lot about tradeoffs. Rarely can you apply a security countermeasure to a system and not trade off convenience, privacy, or something else that users of that system hold dear to their hearts. Bruce Schneier talks a lot about these tradeoffs in real-world systems such as airports (Schneier 2000). In computer systems, the same tradeoffs apply. Forcing users to run with least privilege (as opposed to administrators) is a huge hurdle that many organizations cannot seem to get past, for example, simply because it's painful for users. Most software breaks when run without administrative privileges (which is stupid and should be fixed, as I discuss in WhatIsANonPrivilegedUser).

It stands to reason that when designing secure systems, you should not simply throw random countermeasures at the design, hoping to achieve security nirvana, but you'd be surprised how often this happens. For example, there's something magical about the acronym RSA. Just because your product uses good cryptographic algorithms (like RSA) doesn't mean it's secure! You need to ask yourself some questions.

- Who are my potential adversaries?
- What is their motivation, and what are their goals?
- How much inside information do they have?
- How much funding do they have?
- How averse are they to risk?

This is the start of a threat model. By sitting down with a small group of bright people who span a product's entire life-cycle (product managers, marketing, sales, developers, testers, writers, executives), you can brainstorm about the security of that product. Once you figure out the bad guys you're up against (Schneier 2000 has some great guidance here), you can start to think about the specific threats to your system. Now you'll be asking questions like these:

- Is my system secure from a malicious user who sends me malformed input?
- Is my database secure from unauthorized access?
- Will my system tolerate the destruction of a data center in a tactical nuclear strike?

I'm not being facetious here. Someone who asserts an unqualified "My system is secure" either is a fool or is trying to fool you! No one can say a system is "secure" without knowing what the threats are. Is your system secure against a hand grenade? Probably not. You can have security theater or you can have real security, and if you want the latter, you'll need to think about the specific threats that you want to mitigate. As you'll see, you'll never be able to eliminate all threats. Even if you could, you'd be eliminating all risk, and businesses rarely prosper without a certain margin of risk. Heck, if you disconnect a computer and bury it in 20 feet of freshly poured concrete, there's very little risk that anyone will steal its data, but accessing that data yourself will be a bit challenging. Real security has a lot to do with risk management, and one of the first steps

to achieving a good balance between threat mitigation and ease of use is to know the threats!

But how can you possibly analyze all the threats in a nontrivial system? It's not easy, and you'll likely never find them all. Don't give up hope, though. Due diligence here will really pay off. Most threat models start with data flow diagrams that chart the system. Spending the time to build such a model helps you understand your system better, and this is a laudable goal on its own, wouldn't you say? Besides, it's impossible to secure a system that you don't understand. Once you see the data flows, you can start looking for vulnerabilities.

Microsoft has an acronym that they use internally to help them find vulnerabilities in their software, STRIDE (Howard and LeBlanc 2000):

- Spoofing
- Tampering
- Repudiation
- Information disclosure
- Denial of service
- Elevation of privilege

Spoofing is pretending to be someone or something you're not. A client might spoof another user in order to access his personal data. Server-spoofing attacks happen all the time: Have you ever gotten an e-mail that claims to come from eBay, and when you click the link, you end up at a site that looks a lot like eBay but is asking you for personal information that eBay would never request (like your Social Security number or PIN codes)? This attack is now so common that it's earned a specific name: phishing.

Tampering attacks can be directed against static data files or network packets. Most developers don't think about tampering attacks. When reading an XML configuration file, for example, do you carefully check for valid input? Would your program behave badly if that configuration file contained malformed data? Also, on the network most people seem to think that encryption protects them against tampering attacks. Unless you know that your connection is integrity protected (WhatIsCIA), you're better off not making this assumption because many encryption techniques allow an attacker to flip bits in the ciphertext, which results in the corresponding bits in the plaintext being flipped, and this goes undetected without integrity protection.

Repudiation is where the attacker denies having performed some act. This is particularly important to consider if you plan on prosecuting an attacker. A common protection against repudiation is a secure log file, with timestamped events. One interesting consideration with these types of logs is the kind of data you store in them. If the log file were to be included in a court subpeona, would it be more damaging to your company to reveal it? Be careful what you put in there!

Information disclosure can occur with static data files as well as network packets. This is the unauthorized viewing of sensitive data. For example, someone running a promiscuous network sniffer such as NETMON.EXE can sniff all the Ethernet frames on a subnet. And don't try to convince yourself that a switch can prevent this!

Denial of service (DOS) is when the attacker can prevent valid users receiving reasonable service from your system. If the attacker can crash your server, that's DOS. If the attacker can flood your server with fake requests so that you can't service legitimate users, that's DOS.

Elevation of privilege allows an attacker to achieve a higher level of privilege than she should normally have. For example, a buffer overflow in an application running as SYSTEM might allow an attacker to run code of her choosing at a very high level of privilege. Running with least privilege is one way to help avert such attacks (WhatIsThePrincipleOfLeastPrivilege).

Another technique that is useful when rooting out vulnerabilities is something called an attack tree. It's a very simple concept: Pick a goal that an attacker might have—say, "Decrypt a message from machine A to machine B." Then brainstorm to figure out some avenues the attacker might pursue in order to achieve this goal. These avenues become nodes under the original goal and become goals themselves that can be evaluated the same way. I show a simple example in Figure 3.1.

Figure 3.1 Building an attack tree

You can continue the analysis by drilling down into each new goal (Figure 3.2).

Figure 3.2 Further developing an attack tree

The beauty of attack trees is that they help you document your thought process. You can always revisit the tree to ensure that you didn't miss something. Entire branches of an attack tree can sometimes be reused in different contexts.

Once you have a list of vulnerabilities, you need to prioritize them. Remember that, just like in business, good security really comes down to good risk management. The simplest way to prioritize threats is with two factors: damage and likelihood. Rate each vulnerability on a scale of one to ten based on the amount of damage a successful exploit might cause (financial damage, reputation damage, or even physical damage to persons or property). Calculate a second rating on the likelihood of someone being able to pull off the attack. To prioritize, calculate the overall risk factor for each vulnerability: Risk = Damage * Likelihood. Sort your vulnerabilities into a list of decreasing risk, and address

the highest risk items first. This is a highly subjective analysis, so you'll be glad you built a well-rounded threat modeling team when it comes time to rank the threats.

Now you need to figure out how you're going to manage the risk for each vulnerability. You've got four choices.

- Accept the risk.
- Transfer the risk.
- Remove the risk.
- Mitigate the risk.

Accepting risk is part of everyday life in business. Some risks are so low and so costly to mitigate that they may be worth accepting. For example, accepting the threat of a nuclear strike that destroys two data centers in two different locations simultaneously might just be the cost of doing business. But each threat model is different: For a military system, this might be worth mitigating.

Transfer of risk can be accomplished many ways. Insurance is one example; warnings are another. Software programs often transfer risk to the users via warnings. For example, try enabling Basic Authentication in IIS and you'll be warned that passwords will be sent in the clear unless you also enable SSL.

Remove the risk. Sometimes after analyzing the risk associated with a feature, you'll find that it's simply not worth it and the feature should be removed from the product. Remember that complexity is the number-one enemy of security. In many cases this simple approach is the best.

Mitigating a risk involves keeping the feature but reducing the risk with countermeasures (WhatIsACountermeasure). This is where designers and developers really need to be creative. Don't be surprised if this means reshaping the requirements, and perhaps the user's expectations, to allow the feature to be secured.

Threat Modeling (Swiderski and Snyder 2004) is an entire book dedicated to threat modeling. The second edition of *Writing Secure Code* (Howard and LeBlanc 2002) also has a chapter dedicated to this topic, and *Secrets and Lies* (Schneier 2003) is an invaluable resource as well. To learn more about social engineering, check out *The Art of Deception* (Mitnick 2002). And the next time someone claims that her feature is secure, ask to see her threat model!

1.4 What is the principle of least privilege?

The principle of least privilege was originally defined by Saltzer (1975):

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. . .

I sometimes like to think about this principle in reverse. Imagine if you ignore it entirely and run all your code with full privileges all the time. You've basically turned off a whole raft of security features provided by your platform. The less privilege you grant to a program, the more walls are erected around that program by the platform. If a program misbehaves because of a coding error, perhaps one that was tickled by a malicious user providing malformed input (WhatIsSecureCode), you'll be glad those walls are in place.

Security compromises usually occur in stages: The attacker gains a certain level of privilege via one security hole and then tries to elevate his privilege level by finding another hole. If you run programs with more privilege than they really need, the attacker's life is much easier.

This principle can be applied in many different places; it really is a mindset that you should follow as you design and build systems. The following paragraphs describe some examples that are relevant to .NET programmers working on Windows. The examples are by no means exhaustive, but they will give you some concrete ideas that will help you get into the right mindset.

Daemon processes on servers should be designed and configured to run with only the privileges they need to get the job done. This means that you should absolutely avoid the SYSTEM account when configuring the ASP.NET worker process, Windows Services, COM+ servers, and so on (HowToChooseAnIdentityForADaemon).

Desktop applications should be designed to conform to the Windows Logo guidelines[1] to ensure that they don't attempt to write to protected parts of the file system or registry. When you ship programs that don't follow these guidelines, they break when users attempt to run with least privilege (under normal, nonadministrative user accounts). If you don't want your Mom browsing the Web as an administrator, then start writing programs that she can use as a normal user (WhatIsANonPrivilegedUser)!

When opening files or other secure resources, open them only for the permissions you need for that session. If you plan on reading a file, open it for read-only permissions. Don't open it for read-write permissions thinking, "Someday I may want to write to that file." Open resources for the permission you need at that particular moment.

Use the least privileged form of state management you can for your application. In the .NET Framework, storing application state via Isolated Storage requires less privilege than using a named file, and it has the added benefit of ensuring that your data is written to the user profile (WhatIsAUserProfile), which is one of the Windows Logo guidelines I alluded to earlier.

Close references to files and other resources as soon as possible. This is especially important if you use impersonation (WhatIsImpersonation), as these resources can "leak" from one security context (WhatIsSecurityContext) to the next if you're not careful. Remember that Windows and the .NET Framework tend to check permissions only when resources are first opened. This is a performance enhancement, but it can also lead to security holes if you don't close those resources promptly when you're finsished using them.

And, finally, choose to run with least privilege whenever you log in to your computer, whether you're at home or at work. HowToDevelopCodeAsANonAdmin provides tips for software developers trying to run with least privilege.

[1] http://www.microsoft.com/winlogo

1.5 What is the principle of defense in depth?

During the Cold War, the United States wanted to learn more about Soviet submarine and missile technology. How fast were the Soviets progressing? What were the results from their ICBM tests? Even more important, were the Soviets working toward a first strike capability? In October of 1971, the United States sent its most advanced nuclear spy submarine, the USS *Halibut*, deep into Soviet territory in the Sea of Okhotsk. It's mission? Find the undersea telephone cable that connected the Soviet submarine base at Petropavlovsk to the Soviet Pacific Fleet headquarters on the mainland at Vladivostok (Figure 5.1). The mission was a success, and you can imagine the mood of the divers as they eavesdropped on the wire with an instrument that measured electromagnetic emanations. What they heard was easily understandable Russian conversations—no encryption. The following year, the *Halibut* installed a permanent tap on the line to record the conversations, with a plan to return in about a month to retrieve the records. Eventually more taps were installed on Soviet lines in other parts of the world—the more advanced instruments could store a year's worth of data. All in all, the intelligence gathered from these exercises helped end the Cold War, as it gave the United States a window directly into the Soviet mind (Sontag and Drew 1998)

Figure 5.1 The Sea of Okhotsk

What does this story have to do with computer security? It demonstrates what can happen when systems are designed without redundant security measures. The Soviets assumed that their conversations were secure simply because they were being carried on phone lines that were protected by perimeter defenses (the entrance to the Sea of Okhotsk is much more narrow than my map might first indicate and could easily be defended by the Soviet Navy).

Companies rely on firewalls for perimeter security. Most developers assume that if they are behind the firewall they're safe. But think how easy it is for an attacker to slip behind a firewall. Want some information from an internal company database? Just pick up the phone and call someone in the company and ask for it (Mitnick 2002). Or use a modem to contact someone's computer inside the company. Or park your car in the company's underground parking garage and surf onto the company intranet via an insecure wireless

connection set up by the employees for convenience. Feeling cocky? Walk into the company's headquarters and ask if you can do some work in an empty conference room while you wait for an employee to meet you. Then plug into the Ethernet jack in the room and party on. You'll be surprised how far you get once you find out the names of a few employees. Don't want that much risk? Then compromise an employee's home computer and use his VPN connection to access the network.

These examples assume you are worried only about outsiders getting behind the perimeter. What about the employees who are authorized to work behind the firewall each and every day? I'm not trying to insult you or your colleagues, but you should know that your fellow employees aren't always thinking about the good of the company when they're on the job.

Defense in depth is all about building redundant countermeasures into a system. Don't assume a perimeter defense will protect you. Don't assume someone else's code will protect you. When you write a component, validate the input assuming it can be purposely malformed. Just because your component is currently used only by other trusted components doesn't mean those other components were written by people who know how dangerous malformed input can be (WhatIsSecureCode). Besides, components are designed for reuse, so a component that's used by trusted components today might be deployed in a less trusted environment tomorrow. And never assume that because you're behind a firewall that your internal network conversations are automatically secure. Make sure internal server to server communication is protected (WhatIsCIA).

Always think about failure. Secure systems should not fail badly; rather, they should bend and flex before they break (Schneier 2000). Using a mixture of countermeasures that overlap can provide a synergy that makes the resulting system more secure than its individual parts (WhatIsACountermeasure).

1.6 What is authentication?

Authentication answers the question Who are you? When Alice logs in to a machine, the machine challenges her to prove her identity by asking for a password (or something else, like a smartcard). This is one example of authentication. Another is when Alice is already logged in to a machine and requests a file from another machine via a file share. Even though she's already logged in to the local machine, that doesn't help the remote machine at all. The remote machine wants proof that this is really a request from Alice as opposed to some attacker on the network just pretending to be her. Kerberos (WhatIsKerberos) is an example of a network authentication protocol that protects most Windows systems.

It sometimes helps to break down the question Who are you? into three, more specific questions.

- What do you have?
- What do you know?
- What are you made of?

Asking one or more of these questions can help you infer the identity of a user. For example, a password is something that only that user should know, whereas a smartcard raises two questions: What do you have (the card) and What do you know (the PIN code on the card). This is referred to as multifactor authentication and can lead to considerably more secure systems. The last question queries biometric data such as hand geometry, retinal patterns, thumbprints, and so on. (Schneier 2000) talks a bit about the pros and cons of biometrics, pointing out that they often aren't all they are cracked up to be. For example, thumbprint readers have been shown to be incredibly easy to fool (Smith 2002).

Network authentication can happen in one of three ways. The server can ask the client to prove her identity (the default mode in Kerberos). The client can ask the server to prove his identity (the default mode in SSL), or we can have mutual authentication, where both client and server are assured of each other's identities (both Kerberos and SSL support this as an optional mode). Usually you should prefer mutual authentication wherever possible, unless anonymity is an important feature of the service you happen to be providing. I find it interesting that, in many cases where it seems as though you're getting mutual authentication, you really aren't. For example, when you log in to a Web server over SSL by typing a user name and password into a form, logically you think you've established mutual authentication. You've proven your identity with a user name and password, and the server has proven its identity with a certificate and its private key. But did you double-click the lock to actually look at that certificate? Did you look closely at the URL in the browser address bar? Probably not. For all you know, the server is being spoofed by a bad guy who has simply duplicated the look and feel of the real server. The same problem exists in some of the built-in security mechanisms in Windows. For example, COM has always claimed to use mutual authentication. But the dirty little secret is that, unless you set up a server principal name (WhatIsAServicePrincipalName) and specify it in the client code, you're not really authenticating the server; you're just trusting the server to tell you whom it's supposed to be running as. But nobody does this, just as