Threat Modeling

Uncover Security Design Flaws Using The STRIDE Approach

Shawn Hernan and Scott Lambert and Tomasz Ostwald and Adam Shostack

---

This article discusses:

- The importance of threat modeling
- How to model a system using a data flow diagram
- How to mitigate threats

This article uses the following technologies:
STRIDE

Contents

W
hether you're building a new system or updating an existing one, you'll want to consider how an intruder might go about attacking it and then build in appropriate defenses at the design and implementation stages of the system. At Microsoft, we approach the design of secure systems through a technique called threat modeling—the methodical review of a system design or architecture to discover and correct design-level security problems. Threat modeling is an integral part of the Security Development Lifecycle.

There are multiple approaches to threat modeling, and anyone who tells you his method is the only right one is mistaken. There aren't any well-established ways to measure the quality of a threat model, and even the term "threat" is open to interpretation. Of course that's the nature of the beast; even in the more mature field of cryptography, many popular algorithms have not been proven to be secure. But, while we can't often prove that a given design is secure, we can learn from our mistakes and avoid repeating them. That is the essence of threat modeling.

In this article we'll present a systematic approach to threat modeling developed in the Security Engineering and Communications group at Microsoft. Like the rest of the Security Development Lifecycle, threat modeling continues to evolve and to be applied in new contexts. As you create your own processes for developing secure code, this approach might serve you well as a baseline.

Designing Secure Software

It's difficult enough to design good software, and security makes it even tougher. Flaws that are embedded in a system may or may not be encountered during ordinary use. Indeed, under ordinary use some flaws don't really matter. But in a security context, flaws do matter because attackers can induce failures by setting up the highly specific conditions necessary to trigger a flaw. Something that may have only a one in a billion chance of happening randomly might be dismissed as irrelevant. But if that flaw has security implications, you can bet that an attacker will take advantage of it.

One of the problems in designing secure software is that different groups think of security in different terms. Software developers think of security primarily in terms of code quality while network administrators think of firewalls, incident response, and system management. Academics may think of security mostly in terms of the classic Saltzer and Schroeder design principles, security models, or other abstractions. Of course, all of these things are important in building secure systems. See Figure 1 for a summary of Saltzer and Schroeder's design principles. But maybe the single biggest problem is a lack of security success criteria. If we want to avoid security failures, it means we have to have some idea of what security success looks like.

Figure 1 Security Design Principles (Close Figure 1)

Principle	Explanation
Open design	Assume the attackers have the sources and the specs.
Fail-safe defaults	Fail closed; no single point of failure.
Least privilege	No more privileges than what is needed.
Economy of mechanism	Keep it simple, stupid.
Separation of privileges	Don’t permit an operation based on a single condition.
Total mediation	Check everything, every time.
Least common mechanism	Beware of shared resources.
Psychological acceptability	Will they use it?

Fortunately, we do have some idea of what security means. It means that the systems have the properties of confidentiality, integrity, and availability, that users are authenticated and authorized correctly, and that transactions are non-repudiable. Figure 2 explains each property. You want your systems to have all of these properties, but there is no integrity or availability API. What do you do?

Figure 2 Security Properties (Close Figure 2)

Property	Description
Confidentiality	Data is only available to the people intended to access it.

Integrity	Data and system resources are only changed in appropriate ways by appropriate people.
Availability	Systems are ready when needed and perform acceptably.
Authentication	The identity of users is established (or you're willing to accept anonymous users).
Authorization	Users are explicitly allowed or denied access to resources.
Nonrepudiation	Users can't perform an action and later deny performing it.

Threat Modeling and STRIDE

One way to ensure your applications have these properties is to employ threat modeling using STRIDE, an acronym for Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. Figure 3 maps threats to the properties that guard against them.

Figure 3 Threats and Security Properties (Close Figure 3)

Threat	Security Property
Spoofing	Authentication
Tampering	Integrity
Repudiation	Non-repudiation
Information disclosure	Confidentiality
Denial of service	Availability
Elevation of privilege	Authorization

To follow STRIDE, you decompose your system into relevant components, analyze each component for susceptibility to the threats, and mitigate the threats. Then you repeat the process until you are comfortable with any remaining threats. If you do this—break your system down into components and mitigate all the threats to each component—you can argue that the system is secure.

Now, the truth of the matter is that we can't prove it. We can't yet show that the interactions between components that are individually immune to a spoofing threat aren't susceptible to a spoofing threat when they're composed into a system. In fact, frequently threats materialize only when systems are joined to create larger systems. In most of those cases, the very act of combining subsystems into larger systems involves violating the original assumptions the subsystem made. If a system was never designed to be used over the Internet, for example, when you expose it to the Internet, new security concerns will emerge.

In any case, what is true is that if any component of the system is susceptible to a spoofing threat, you can't say that all your users are properly authenticated.

Data Flow Diagrams

Data flow diagrams (DFDs) are typically used to graphically represent a system, but you can use a different representation (such as a UML diagram) as long as you apply the same basic method: decompose the system into parts and show that each part is not susceptible to relevant threats.

DFDs use a standard set of symbols consisting of four elements: data flows, data stores, processes, and interactors, and for threat modeling we add one more—trust boundaries. Figure 4 shows the symbols. Data flows represent data in motion over network connections, named pipes, mail slots, RPC channels, and so on. Data stores represent files, databases, registry keys, and the like. Processes are computations or programs run by the computer.

Figure 4 DFD Symbols (Close Figure 4)

Item	Symbol
Data flow	
Data store	
Process	
Multi-process	
Interactors	
Trust boundary	

Interactors are the end points of your system: the people, Web services, and servers. In general, they are the data providers and consumers that are outside the scope of your system, but clearly related to it.

Trust boundaries are perhaps the most subjective of all: these represent the border between trusted and untrusted elements. Trust is complex. You might trust your mechanic with your car, your dentist with your teeth, and your banker with your money, but you probably don't trust your dentist to change your spark plugs.

Getting the DFD right is key to getting the threat model right. Spend enough time on yours, making sure all the pieces of your system are represented. Have you noted all the files and registry keys your app touches? Are you reading data from the environment?

Each of the elements (processes, data stores, data flows, and interactors) has a set of threats it is susceptible to, as you see in Figure 5. This chart, along with your DFD, gives you a framework for investigating how your system might fail. It's probably a good idea to parcel out the investigative work to subject matter experts and build checklists to ensure you don't make the same mistake twice. For example, you could have your networking team investigate how information disclosure threats apply to your network data flow. They will understand the relevant technologies and be well suited to do the research on security as it pertains to their portion of the application.

Figure 5 Threats Affecting Elements (Close Figure 5)

Element	Spoofing	Tampering	Repudiation	Information Disclosure	Denial of Service	Elevation of Privilege
Data Flows		"		"	"	
Data Stores		"		"	"	

Processes	"	"	"	"	"	"
Interactors	"		"			

A Sample System

Let's say you need a system to collect the accounting files from your sales force, compute sales data on your database server, and produce weekly reports. We'll call the system the Fabrikam analyzer database. The goal is fairly simple: getting files from a set of systems and performing some analysis of the files on a centralized server. It's the business goal as the customer stated it, but you need to turn the problem statement into specifications and plans.

There are many obvious potential threats to this system, and many of them come from the implicit security requirements of the problem statement. The collection process alone raises a number of questions. Collecting information means moving it from one place to another. How are you going to secure it in transit?

You'll be manipulating accounting files, which by their very nature are sensitive and often subject to legal requirements. And you'll need to identify a specific group of people—the sales force. How will you know them? The problem statement implies a number of security requirements:

- The data has to be protected from inadvertent disclosure in transit and in storage.
- The sales force needs to be authenticated and authorized.
- The application must respond gracefully to attacks that rely on malicious input, like SQL injection and buffer overflows.
- The server needs to be able to perform the calculations at least weekly.

Customers may never state these explicitly, so designers must find the security requirements inherent in the problem statement.

Of course, there are also lots of less obvious security requirements that need to be addressed. What happens if the files can be overwritten on the server in such a way that a salesperson could conceal a suspicious sale? What happens if one salesperson can claim credit for the sales of another?

And just who is "we"? What if Contoso Corporation is able to pose as the Fabrikam server to the Fabrikam sales force? Remember, the attacker doesn't have to respect your protocol or use your tools to access your servers; he will spend more time than you imagine looking for flaws and he may well have a large number of machines available for performing complex calculations.

At this point, if the component is a low-risk system or you have a great deal of experience in similar systems, you may decide that it's reasonable to go off and start planning mitigations for the threats. And that's OK. But what if it's a high-risk component? How do you know what you don't know?

If you stop at this point, your threat model will be limited not only by what you know, but what you happen to remember at the time you're working on it. You not only have to think like an attacker, you have to think like all attackers. Simultaneously.

Applying STRIDE to the Fabrikam Analyzer Database

Now, let's try to create a data flow diagram from the problem statement. An initial attempt might look like Figure 6. On the server side, there are two processes, two data stores, and three data flows. There is one trust boundary between the server and client sides of the system, with one data flow that crosses the trust boundary for each client. For each client, there is a salesperson, an accounting application, the accounting data, and the sending process.

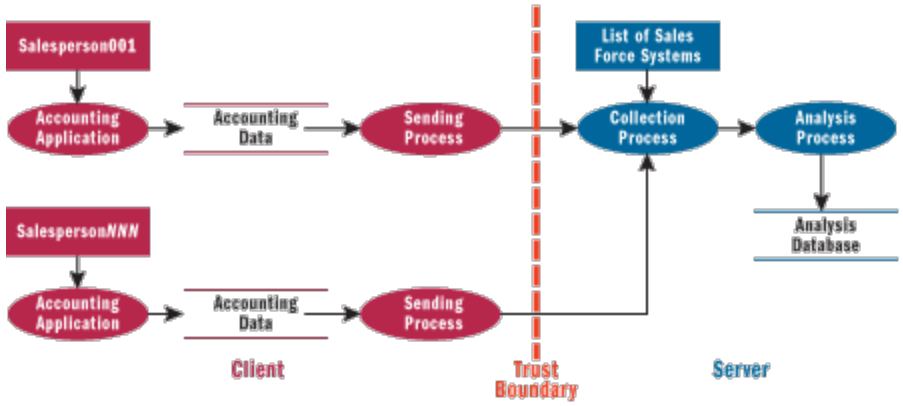

Figure 6 An Initial DFD for the Analyzer Database

But is this the right DFD? Is this the DFD that will yield the most complete picture of threats? Some simple rules of thumb suggest that it's probably not the right one. To begin with, there is a data sink. Data goes into the Analysis Database and never gets read. The customer never mentioned anything about reading the data explicitly because it wasn't related to the problem at hand. A designer might say something like "that part of the problem is out of scope." But from a security point of view, it's not out of scope at all. So you need to represent the reader of the data somehow.

Here are some general rules for understanding if your DFD is sensible. First, be careful of magic data sources or sinks: data isn't created out of thin air. Make sure you have a user represented as a reader or writer for each data store. Second, beware of psychokinesis as a data transport. In other words, make sure there is always a process that reads and writes data. It doesn't go directly from a user's head to the disk, or vice versa. Third, collapse similar elements within a single trust boundary into a single element for modeling purposes. If they are implemented in the same technology and are contained within the same trust boundary, you may be able to collapse them. Fourth, be careful when modeling details on either side of a trust boundary. The temptation is to model things on both sides of a trust boundary simultaneously. It's good practice to have a context DFD and breakout diagrams that show more detail. Our system here does represent the client and server systems simultaneously in a single model. But remember that the attacker is under no obligation to use your tools or respect your protocols.

Revising the data flow diagram to take these rules into account, the DFD in Figure 7 is probably a better representation. The changes here are notable. We've collapsed the collection and analysis processes into one. This doesn't necessarily mean that they will be implemented together—the point is to not lose the forest for the trees. Think "major function," not implementation.

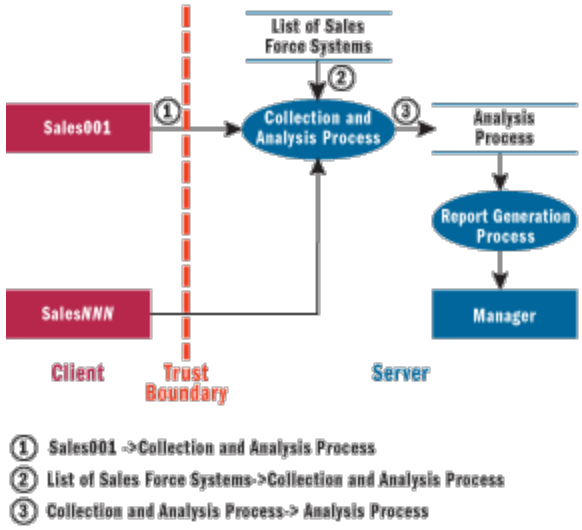

Figure 7 A Better DFD

We've also represented the client side as nothing more than external interactors, and that is a powerful change. This reflects the truth that an attacker is free to do whatever he or she chooses. Consequently, we've limited this threat model to the server side only—a complete threat model would include a similar representation of the client side systems, showing the server as nothing more than a set of external interactors. That's the very definition of a trust boundary—you don't trust what's on the other side.

Now we can represent all this in lists and tables and start breaking the large problem into a series of smaller problems.

Analyzing Data Flows and Data Stores

Let's first look at the data flows. There are three of them as you saw in Figure 6. Each of these data flows is subject to threats listed in Figure 3. Let's see how each process is vulnerable.

Data Flow 1: Sales to Collection Where sales data is transferred to the collection process, tampering is possible. In other words, data might be modified as it travels over the Internet, especially if the data comes from laptops in a variety of security settings. Information disclosure is another risk. Data in transit may be read by those who should not have access.

The collection process might also fall victim to denial of service attacks; an attacker might prevent the collection server from being accessible to the sales people. (If the attacker does this on the last day of a quarter, it could have a material impact on the company.)

Data Flow 2: Sales System List to Collection Similar threats exist for sales system list collection. Someone could tamper with the data by inserting a new system into the list of salespeople that could allow the input of false data. Removal of a system could prevent a salesperson from being able to register sales. (This could be modeled as a denial of service. We call it tampering. Don't get too hung up over the terminology.)

As you consider threats to this data flow, it may occur to you that the authentication system for the sales force has not been identified. An inadequate authentication system would present a spoofing

threat to the sales force. Make sure the threat gets recorded and addressed. The same threat will occur to you when you examine spoofing threats against the sales force. A little redundancy is great.

The list of salespeople is probably interesting to Fabrikam's competitors. It could also be interesting to insiders if layoffs are happening, so the threat of information is significant and needs to be addressed. Don't be lulled into complacency because you can't imagine why someone would want your data. You simply have to assume someone will.

Depending on how large Fabrikam is, and how often the sales force changes, denial of service may not be very important. It's OK (and even sensible) to perform some level of risk analysis when looking at threats. But remember that the further away you are from your customer, the harder it is to know what the customer's tolerance is for different risks. Don't assume too much about your customer's situation or tolerance for risk.

Data Flow 3: Analysis Process to Analysis Store Here we encounter an interesting situation regarding tampering. We have a data flow contained entirely within a trust boundary. In this situation, a hardcore security theorist might say there's absolutely no need to worry about processes entirely within a trust boundary—after all, you trust them.

On the other hand, however, a hardcore security practitioner might reply that anything can fail, and this isn't any different. We have sympathy for both views and would resolve the issue in terms of overall priority. Data flow 1 is clearly more exposed than data flow 3, and the effort you spend examining data flow 1 ought to reflect that greater level of exposure. But data flow 3 isn't entirely without risk.

For example, what if the analysis database is stored on another machine? The threat model as written seems to suggest that the collection process is housed on the same machine as the database, but perhaps it's not. Perhaps the modeler made an incorrect assumption, or the decision hasn't yet been made.

If it turns out that the analysis database is stored on a remote machine, it's very likely that data flow 4 has a similar threat profile to data flow 1. Finally, even if you can rely on the trust boundary, don't forget that situations change. Defense in depth may be a worthwhile investment.

Information disclosure and denial of service are also threats here. In fact, the situation is the same as with the tampering threat. In other words, is the trust boundary reliably and correctly set? If not, these threats need to be considered.

Now let's look at the data stores.

Data Store 1: Analysis Database This data store is vulnerable to tampering, especially because what's in the database hasn't been fully specified. Is this a sales database that includes order fulfillment and salesperson bonuses? Is it a forecasting database that produces sales predictions? Different data types attract different attackers. Note that attackers may be insiders, not outsiders. They may have legitimate access to the database in order to do their jobs.

Information disclosure is a growing problem. If Fabrikam takes sales orders from consumers on credit, there may be Social Security Numbers or credit card numbers in the database. There is probably proprietary information in the database that competitors would like to see. If Fabrikam is providing health or medical supplies, it may be subject to HIPAA privacy constraints.

Denial of service may be a threat as well. Who uses this database, and for what? Filling up the database is a simple attack. When the database is full, common responses are to stop servicing new requests or to overwrite old data. Denial of service attacks matter most when they prevent business goals from being met.

Data Store 2: List of Sales Systems Tampering is a threat here. Attackers could add salespeople who can make mischief or delete salespeople and prevent them from getting their jobs done. Information disclosure and denial of service are also viable threats.

Data Store 3: Laptops Tampering with laptops may allow an attacker to steal data or access control tidbits such as passwords, keys, or certificates. An attacker could install spyware to give himself ongoing access to the system. This probably crosses threat models into other projects, but if you're designing a system that can't reasonably be managed, that may make your customers or users less secure. All information on a laptop is fair game if the attacker controls it. Again, threats here probably involve more than just this part of the threat model.

Analyzing Processes

A process that pretends to be the collection process (spoofing) can collect all of the data that sales is trying to submit. This leads to information disclosure and denial of service. There are many threats that, when realized, can lead to other threats. If you can pretend to be the administrator, for example (spoofing), you can shut down the system (a denial of service threat). Also, if an attacker tampers with the data here and corrupts or reads memory, he may be able to cause a denial or service or elevation of privilege. Don't get too caught up in trying to track down all the follow-up consequences, though.

Repudiation, to which the collection process is vulnerable, is about lying. A repudiation threat is when you can take an action and plausibly deny having taken it. In this case, repudiation threats might involve, for example, submitting revised sales data that overwrites existing data and being able to claim that you did not do it.

Information disclosure threats against a process can be tricky—you normally think of these threats as affecting data flows and data stores. But a process holds lots of valuable data in memory. If an attacker can read from the memory of a process, he might not have to break into the database.

Similarly, if an attacker discovers something about the internal structure of a program, it can help him conduct other types of attacks. An information disclosure attack that allows an intruder to discover the memory address of certain variables (for example) can be a very valuable stepping stone in the attack process.

Both denial of service and elevation of privilege are also quite possible here. Insiders and competitors might be motivated to prevent the ordinary collection of the data, and this could be performed in a variety of ways. For example, an attacker who can cause the collection process to run code they have written can spoof, tamper, or deny service. The collection process is probably exposed to the Internet, and proper secure code will be very important.

You can continue this sort of analysis for each of the threats and each of the elements in the DFD until you can say that you've addressed tampering, information disclosure, and denial of service against all the data flows and data stores; each of the six STRIDE threats against all of the processes; spoofing

and repudiation threats for all interactors; and unique threats that affect your trust boundaries. None of this will guarantee that the system is secure, but it will certainly help you sleep better at night.

Mitigating the Threats

The ideal situation is to mitigate a threat with a strong, well-understood solution. For example, using strong cryptography appropriately is believed to be a strong countermeasure to many types of information disclosure threats. You may never be able to prove that a defense is perfect. However, one of the nice things about the STRIDE model is that it gives you insight into the nature of the mitigations you need. Simply recasting Figure 3 in terms of available technologies gives you an idea of what kinds of mitigations are necessary. Choosing a technology can be challenging, however. In general, I've found that asking two simple questions can be helpful: can the technology be used to mitigate the threat, and would it actually be used in the scenario you're concerned with?

Finding Manifestations of Threats

Now that you have broken down the big problem into smaller problems, how do you solve those smaller problems? You've got some research to do. You want to find previous failure modes for the technologies you'll be using. If a data flow, for example, will be using Remote Procedure Call (RPC) over TCP, you need to find not only general classes of failures in data flow but also specific known failures for TCP and RPC.

One excellent approach to learning how threats manifest themselves is using Chapter 22 of *The Security Development Lifecycle*, by Michael Howard and Steve Lipner (Microsoft Press®, 2006) in which threat trees are developed for STRIDE threats against each of the four standard DFD elements. For example, one threat tree explores how tampering might manifest itself against a data flow in a general sense. The trees are presented such that the leaf nodes of the trees suggest attacks that will realize the threat. The idea is to use the leaf nodes of the threat trees as leading questions—sort of augmenting the brainstorming you did earlier.

So, for example, let's say there is a tampering threat against the Sales to Collection data flow. Looking at the list of questions from *The Security Development Lifecycle*, the first one related to tampering with a data flow is the following: is the data flow defended (hashed, MAC'd, or signed) using anti-replay defenses such as time stamps or counters?

This suggests a simple attack—replaying valid messages—that has thus far been overlooked in the entire discussion! It's not that the attack is novel or previously unknown. It's just that it wasn't obvious in the problem statement, and it would be an easy attack for even seasoned security folks to overlook.

The impact of the attack in this scenario is this: what if a salesperson could submit a set of sales data twice? Imagine that Vishal and Eric are competing for a trip to Hawaii. Eric might be tempted to submit a set of data twice to make his sales numbers look really good and win the trip.

You might think the accounting system would catch that, because after all, that's part of the purpose of an accounting system. But how do you know? This is just a reporting system after all, and maybe the data generated from this system isn't ever double-checked against the official accounting data.

Maybe the attack is mitigated somehow and maybe it isn't. But understanding whether an attack is possible and uncovering attacks that haven't been thought of before are part of the main goal of threat modeling.

Given this newly uncovered threat, you are now presented with a new task: understanding whether the threat has been mitigated. In the end, a likely result of an investigation like this would be that the files are uniquely identified with what is effectively a random number. Now you have a business decision to make: are you comfortable with the risk, or do you need a stronger solution?

The use of prebuilt threat trees has proven effective in a number of situations at Microsoft to ensure that known attacks aren't overlooked. And as new attacks are discovered, the threat trees themselves can be augmented with these new threats. In this way, the threat trees can form a sort of institutional knowledge. Remember, though, that they are very general in nature. There are whole classes of attacks that apply to one technology and not another technology.

In the case of a data flow, for example, how would you know that the security properties of TCP are different from the security properties of User Data Protocol (UDP), and that factors like the initial sequence number and window size play a role in the security of your network connection? If you're familiar with network security, you might recognize these problems; if not, you may overlook these concerns when designing a solution. That is where attack patterns come in.

Attack Patterns

Attack patterns are the manifestation of one or more threats in the context of some specific technology. For example, the strcpy operation in C might permit an attacker inputting long data strings to corrupt the memory of a target program, allowing him to execute code of his choice. This is how a buffer overflow can pose an elevation of privilege threat. Of course, an attacker might use the buffer overflow to do other things as well—tamper with paycheck data for instance. Sometimes people talk about attack patterns in more formal terms, expressing them as a set of preconditions and postconditions. But for our purposes here, they are the specific things that can allow an attacker to realize a threat.

You may or may not be familiar with these kinds of attacks, but in fact there are many more types of attacks that affect specific technologies in specific contexts, and there's little that can be done about it. You have to be familiar with the attacks that affect your technology and build in appropriate defenses.

Conclusion

Designing secure software can be difficult, but as with any challenge, a good strategy is to break down the problem into smaller parts that are more easily solved. For high-risk activities, we believe it is helpful to have an organized framework for doing so, and using the STRIDE model with threat modeling is one such approach. We have used this model in a number of teams within Microsoft and the results have been promising—in nearly every case, we turned up a design issue that might have gone unnoticed until much later. Once you get used to applying the STRIDE model, it often boils down to getting the DFD right, brainstorming attacks, and reviewing the known checklists. This can go pretty quickly.

As you strive to develop secure software, we recommend threat modeling as a key part of your process, and specifically the STRIDE model presented in this article. But the key point is to find a method that works for you, apply it early in your design, keep in mind that any component can fail, and do the necessary research to ensure you've accounted for known attack patterns.

Finally, design is just one part of building secure software. Executive support, implementation, testing, building and delivering, and servicing and maintenance all play crucial roles in the ultimate security of your systems. You've got to get it all right.

Where to Go for Technical Details

For an excellent overview of many common types of attacks, we suggest Writing Secure Code, Second Edition, by Michael Howard and David LeBlanc (Microsoft Press, 2002). For more technically specific texts, see:

- *Securing Web Services with WS-Security*, by Jothy Rosenberg and David Remy (SAMS, 2004).
- *SQL Server Security*, by Chip Andrews (Osborne McGraw-Hill, 2003).
- *Secure Coding in C and C++*, by Robert Seacord (Addison Wesley, 2005).
- *Building Secure Microsoft ASP.NET Applications* (Microsoft Press, 2006).
- *Security for Microsoft Visual Basic .NET*, by Ed Robinson and Michael James Bond (Microsoft Press, 2003).
- *The Software Vulnerability Guide*, by Herbert H. Thompson and Scott G. Chase (Charles River Media, 2005)

Finally, for a list of common security-related defects, one emerging resource is Mitre's Common Flaw Enumeration project.

If you work frequently in any technologies, it's probably a good idea to incorporate common flaws from those technologies into your own threat trees or checklists. And of course, you should review current and past security advisories from your own organization as well as other organizations who have implemented similar technologies. Microsoft makes security bulletins available at microsoft.com/security. Notable third parties include:

- The CERT Coordination Center
- Security Focus
- Secunia

If you're adding a new feature that your competitors may already have, research their security mistakes as well as their successes. Many vendors and third parties make security bulletins freely available.

Shawn Hernan is a Security Program Manager for Microsoft, currently working on training to help meet the vision of delivering software that is secure by design, secure by default, and secure in deployment. Prior to joining Microsoft, he was the vulnerability team leader at the CERT Coordination Center at Carnegie Mellon University.

Scott Lambert is a Security Program Manager on the Secure Windows Initiative (SWI) team at Microsoft. He owns enhancing the internal security tools, including various fuzzing tools. Leveraging his industry experience, Lambert works to ensure that SWI tools identify the vast majority of vulnerability classes.

Tomasz Ostwald currently works as a Security Program Manager in the Security Windows Initiative team, where he conducts Final Security Reviews of products released by Microsoft.

Adam Shostack is a Program Manager on the Security Development Lifecycle (SDL) team at Microsoft. He owns the threat modeling component of the SDL.