Chapter 6
Microsoft Threat Modeling and Threat Modeling Methodology

Revision 5.0, October 2025.

Objectives

- Learn the STRIDE model of threat classification.
- Develop an understanding of the Microsoft Threat Modeling methodology.
- Understand the Microsoft Security Design Lifecycle and how it relates to STRIDE and threat modeling.

6.1 Background

To turn the tide on their early struggles with security issues plaguing their products, Microsoft introduced threat modeling methodologies as a core component of their efforts. They developed this methodology to give their designers and programmers a conceptual and practical tool that would provide a clear understanding, at design time, of the threats that a system could face.

In January of 2002, Bill Gates sent an email to all Microsoft employees announcing the creation of their Trustworthy Computing (TwC) initiative. One of the early outcomes of this initiative was their Security Development Lifecycle (SDL)[1]. This life cycle includes steps for requirements, design, implementation, verification, and release. Threat Modeling is a key part of the design step in the SDL.

A key enabler for promoting the Threat Modeling methodology was the development of the Threat Modeling Tool, which automated much of the process of identifying threats in a design. In 2004, to make their methodology more widely accessible and understood, Microsoft published a book describing the details and application of this methodology[2]. In 2014, Adam

[1] "Microsoft Security Development Lifecycle (SDL), Security Engineering, Microsoft Corp.
`https://www.microsoft.com/en-us/securityengineering/sdl/`

[2] Frank Swiderski and Window Snyder, **Threat Modeling**, Microsoft Press, Redmond, WA, 2004.

Shostak produced a new book trying to make the methodology and the tool more approachable[3].

In this chapter, we explain the Microsoft Security Design Lifecycle, threat classification system used by Microsoft, and steps performed in their Threat Modeling methodology. We then introduce their Threat Modeling Tool.

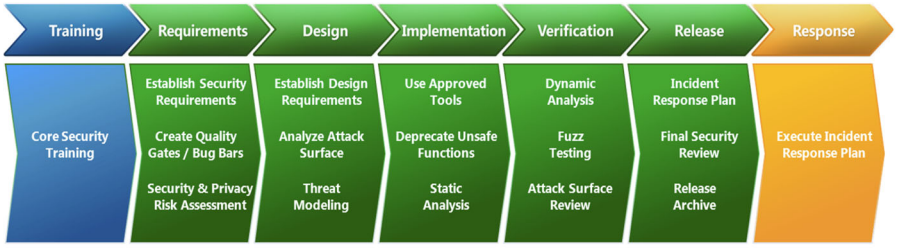


Figure 1: Microsoft Security Development Lifecycle

6.2 Security Design Lifecycle (SDL)

The goal of SDL is to integrate security into each stage of the software lifecycle. This starts even before you consider the design by making sure that your team has the right training and extends beyond the deployment of the software by making sure that you can respond to security incidents as they occur.

The SDL is not a "one and done" methodology. As the software needs to be updated, you will make repeated passes through each stage.

6.2.1 Training

Everyone involved in the project, including the managers, designers, programmers, testers, and release team all need to understand what is their role related to the software security. The training may include formal classes, degree programs, or certificates. Or it may occur on-demand from written or online sources. For release and incident response training, it may involve live exercises and simulations (tabletop exercises).

Each person in each role needs to understand (1) why addressing security risks is important, (2) what they need to do in their role, and (3) how to accomplish these things. The training task is made all the more challenging by the limited skills that are provided during computer degree programs.

---

[3] Adam Shostak, **Threat Modeling: Designing for Security**, John Wiley and Sons, Indianapolis, IN, 2014.

Training is an ongoing process as new threats are always emerging and the technology is constantly evolving. Your organization should have a training schedule to refresh and update the skills of each member.

6.2.2 Requirements

Even before you can write your first functional requirement, you will want to have procedures for how you will design, code, test, and release. For design, that could include a formal threat modeling process. For coding, that could include team coding standards, peer checks on code commits, and compilation standards (such as no warnings allowed). For testing, that could include tool use (such as static analysis and dependency tools), unit and end-to-end testing, and fuzz testing. For release, that can include bug tracking and vulnerability response and remediation.

Your requirements may also include directions on the choice of programming language. For example, are C and C++ allowed or are more modern languages with stronger memory protection required? Similarly, you may include requirements on which programming frameworks (such as for web or database) are suggested or allowed. Besides security, uniformity in use can make code more portable and understandable across different parts of your project or organization.

The above requirements may also be influenced by what kind of data you are handling. Information may include PII (personal identifying information), PHI (protected health information), and PFI (personal financial information).

For example, if your system will be storing PII, there will be a need for increased care and scrutiny of the code. One definition of PII from NIST[4] is:

> *Any information about an individual maintained by an agency, including (1) any information that can be used to distinguish or trace an individual's identity, such as name, social security number, date and place of birth, mother's maiden name, or biometric records; and (2) any other information that is linked or linkable to an individual, such as medical, educational, financial, and employment information.*

There is a similar definition from the European GDPR[5] (General Data Protection Regulation):

> *'Personal data' means any information relating to an identified or identifiable natural person ('data subject'); an identifiable natural person is one who can be identified, directly or indirectly, in particular*

---

[4] https://csrc.nist.gov/glossary/term/PII

[5] https://gdpr.eu/eu-gdpr-personal-data/

by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person.

While the U.S. government does not have a single definition of PII, and not unified regulations for such data, Europe has GDPR, containing a single definition and collection of regulations for European Union member states.

Personal medical information in the U.S. is classified as PHI, defined by the U.S. Department of Health and Human Services as information that[6]:

"(A) is created or received by a health care provider, health plan, public health authority, employer, life insurer, school or university, or health care clearinghouse; and

"(B) relates to the past, present, or future physical or mental health or condition of an individual, the provision of health care to an individual, or the past, present, or future payment for the provision of health care to an individual.

PHI falls under HIPAA (Health Insurance Portability and Accountability Act of 1996)[7] regulations in the U.S., so any software design implementation must account for the requirements specified by these regulations.

PFI is another category of information that may need careful handling. There are multiple laws and regulations around the world describing the requirements for storing this information and handling disclosures when there is a data breach.

Of course, in addition to following all these security requirements, you will want to carefully specify the functional requirements of the software that you will build.

6.2.3 Design

The design phase is critical to the security of any software system. As with any design phase, it starts by an analysis of the threats to the system using some form of threat modeling. Such an analysis is based on a firm understanding of the attack surface (Chapter 1.2). With this understanding, you may have identified weak spots in your system that you can use to help revise the design with the goal of making the attack surface as small as possible.

---

[6] https://www.hhs.gov/answers/hipaa/what-is-phi/index.html

[7] https://aspe.hhs.gov/reports/health-insurance-portability-accountability-act-1996

As you work on your design, you will want to incorporate good design principles, such as the ones presented in the chapter on secure design principles (Chapter 4)

6.2.4 Implementation

The implementation stage is what most of us associate with building software. In fact, few university courses discuss requirements or design. The requirements are handed to you in the form of a programming assignment and the design is just what happens when you start programming. Except for some software engineering or project courses, almost no courses require you to submit a design before you start coding.

Coding standards and careful coding practices are key to a safe and secure implementation. We have several chapters in Module 3 devoted to different aspects of secure programming, showing you the dangers you might encounter and practices needed to avoid such dangers. These practices are worth mastering to improve the code that you develop. Note that these are not a complete list of what you might encounter but are a good starting place.

As part of your implementation process, you will want to use tools that help you find problems in your code and in the code that you include or link against. Module 6 discusses some of these tools and how they work. As we note in Chapter 6.1, a good starting place for tool use is to take advantage of your compiler. Modern compilers use sophisticated code analysis algorithms, both to check for dangerous practices and help generate efficient code. Turning on all the warnings that your compiler provides will give you a head start in detecting dangerous coding practices.

6.2.5 Verification

The verification stage is a chance to catch problems that were not detected during implementation. At this point in the life cycle, you can use dynamic tools, i.e., ones that catch problems while the program is executing. Some examples of such tools include dynamic analysis tools and fuzz testing.

Dynamic analysis tools watch your program execute to try to detect mistakes that the program is making. For example, the Clang Memory Sanitizer[8] adds extra code to your C or C++ program that makes checks on your use of pointers and dynamically allocated memory. It can catch things like use of uninitialized or NULL pointers, use after free of a pointer, or access to an invalid region of memory. There are many tools that can provide similar functionality, such as the open source Valgrind Memcheck[9] or the TotalView

---

[8] `https://clang.llvm.org/docs/MemorySanitizer.html`

[9] `https://valgrind.org/docs/manual/quick-start.html`

Memscape[10] product. Given the memory risks associated with using C and C++, such tools can be considered essential to use.

Fuzz random testing, described in later chapters, offers an easy-to-use way of identifying flaws in the way that your program handles unexpected inputs. At the most basic level, such flaws can cause program crashes. However, such flaws are indicative of a program reaching a state that you, as the programmer, did not anticipate. If such a flaw exists in your code, it might be harnessed to get your program to do something unexpected, creating an opportunity for a vulnerability.

6.2.6 Release

The software development process does not stop when the coding and testing are done. You will need a well organized release process and that process has distinct security implications.

First, you will need a way of organizing security updates along with your functionality updates. Many organizations separate these two streams. The goal is not to force a user to go to a new functionality version of your software just to fix a security vulnerability. Fixing security vulnerabilities is an urgent task that all users should want to do as soon as possible. Upgrading to a new functionality version of the software is something that is often scheduled carefully to avoid incompatibilities and temporary loss of functionality.

Second, you want to have a checklist of security activities that need to be completed before a new version is released. Microsoft calls this a Final Security Review (FSR). An FSR might involve a review of all the tool reports, static and dynamic, to make sure that no serious coding flaws are allowed to be distributed. A review may involve a triage process where pending flaws are categorized as fixed, pending but not serious enough to block a release, or release blockers (and therefore need to be addressed before the release). In addition, you will want to check that no debugging messages stay active in the release version of the software. Such messages could leak private information.

Third, you need to make sure that installation instructions and scripts are updated to account for any security issues. Even the most secure code, if installed incorrectly, can allow unauthorized access to the system. For example, if a file holding credentials is installed in such a way that any user can overwrite it, then the system is likely to be vulnerable.

---

[10] https://totalview.io/products/totalview/leak-detection-memory-debugging

Fourth, for the most critical systems, you may want to undergo a periodic in-depth vulnerability assessment of the code. This is a time consuming and therefore expensive activity, so has to be undertaken judiciously. The First Principles Vulnerability Assessment (FPVA) methodology described in Module 5 is an example of an approach that you can use.

6.2.7 Response

The final stage occurs when the bad news arrives; someone found a vulnerability in the code and you need to respond to it. (Note that we say "when" it arrives, not "if".)

In preparation for this stage, you need to formulate an incident response plan (IRP). Such a plan should:

1. Design the reporting channels for such information. How does a user or team member communicate that a vulnerability has been found? To what outside sources of threat intelligence will you subscribe to have better visibility of emerging threats and vulnerabilities? Many organizations join their sector-specific ISAC (Information Sharing and Analysis Center), which gathers and disseminates such threat information. For example, the mission of the Space ISAC is described as:

   The Space ISAC serves to facilitate collaboration across the global space industry to enhance our ability to prepare for and respond to vulnerabilities, incidents, and threats; to disseminate timely and actionable information among member entities; and to serve as the primary communications channel for the sector with respect to this information.
2. Designate who is responsible for handling vulnerability reports when they arrive. This person or team will have the responsibility for identifying the urgency of the report, disseminating it to the approach technical team, notifying management if this is a serious vulnerability, coordinate with government agencies such the DHS/CISA in reporting and getting help for remediating the issues, notifying users, and working with the press.
3. As mentioned before, have a plan for scheduling an urgent security release of the software. For each currently supported version of the software affected by the vulnerability, there should be a separate security release that remediates it.
4. Coordinate with the software development team to make sure that future releases of the software incorporate any needed changes.

5. File a Common Vulnerability and Exposures (CVE) report to the NIST National Vulnerability Database[11], which is used as a worldwide repository of such information. Such a report will ensure that dependency tools will know that they should to look for the affected versions of the software and report them to users.
6. Communicate to your users the seriousness of the vulnerability, how to detect it, and how to remediate it.

6.3 STRIDE

STRIDE is a threat classification model developed as part of Microsoft's Trustworthy Computing (TwC) initiative[12] (one of the authors of this book assembled the letters of the acronym for the original paper[13]). The model is a useful way to categorize common threats to a software system.

Note that STRIDE has broader applicability than the Microsoft Threat Modeling methodology and is sometimes used separately or incorporated into other threat modeling methodologies. When filing a security vulnerability report, noting the STRIDE category is a quick and easy way to characterize the issue for others to quickly see.

The six STRIDE threat categories are as follows, each to be explained in detail with examples following:

Spoofing of an identity
Tampering
Repudiation
Information disclosure
Denial of service
Elevation of privilege

STRIDE aids threat modeling as a kind of checklist of the types of threat to consider, though it is important to bear in mind that it may not necessarily

---

[11] `https://nvd.nist.gov/`

[12] Aanchal Gupta, "Celebrating 20 Years of Trustworthy Computing", Microsoft Corp., January 2022.
`https://www.microsoft.com/en-us/security/blog/2022/01/21/celebrating-20-years-of-trustworthy-computing/`

[13] Kohnfelder, Loren; Garg, Praerit: "The threats to our products". *Microsoft Interface* (April 1, 1999).
`https://adam.shostack.org/microsoft/The-Threats-To-Our-Products.docx`

cover every possible threat to every system. For example, when looking at a system component block diagram or reviewing the security of a protocol, it helps to write out the six letters to make it easy to remember the corresponding threat categories, and then apply each to the analysis at hand for separate detailed consideration. For example, S: Is there an identity-spoofing threat here and what are the implications of that spoofing? T: What data might be subject to tampering and what are the implications of that tampering? We can ask similar questions for R, I, D, and E.

Some threats will involve more than one category and, generally, as long as you identify one of them and mitigate it, you often get two birds with one stone. For example, if an attacker spoofs the identity of an administrator, they also acquire the associated high level of access, which is an elevation of privilege. These generic categories are defined to help, not to make the job harder, but if you can identify these overlapping cases it helps later assessing the effectiveness of mitigations.

Note that repudiation (R) is different from the other five threat categories as it does not involve an action related to attacking the system. Instead, it is an independent action to provide deniability of proof that you committed some inappropriate operation on a system.

STRIDE has its limitations in that it does not capture all the kinds of bad things that might result from an attack. For example, if you were creating a threat model of a medical device, a particular configuration error might cause a malfunction creating a dangerous condition for the patient that could harm their physical health. There is no STRIDE category that captures this concept of physical harm to the user. Perhaps we need a new acronym such as STRIDED, where the last "D" could stand for "damage" or "danger".

6.3.1 Spoofing

Spoofing of an identity is when a person or program successfully impersonates another, thereby concealing the attacker's true identity and often gaining unauthorized access in the process.

- In our vault example (from Chapter 2), an attacker impersonates Rich Customer and convinces the bank to allow them to have access to Rich Customer's money.
- Sender email address spoofing.
- Man-in-the-middle intrusion in communication such as impersonating a web service.
- Web page referrer spoofing.

6.3.2 Tampering

Tampering includes unauthorized modification or deletion of data or code that alters the system behavior.

- In our vault example, Evil Attacker manipulates the security cameras so the attacker can access the safe without being seen.
- Installing backdoor access (a secret, unauthorized access path). Note that this might also be thought of as an Elevation of Privilege as the backdoor provides increased access.
- Disabling security monitoring.
- Subverting authorization.
- Bypassing valid license checks.
- Altering control flow.
- Injecting malicious code into a program.
- Changing the balance amount of an account.

When tampering occurs, it can be much more pernicious if it goes unnoticed for a long period of time. In a commonly used devious form of attack, code is left behind (often termed an **implant**) that may be hidden or intentionally left inactive for a period of time, making it difficult to detect. Persistent implants can survive reboots and system updates and can be very difficult to eradicate. Similarly, subtle tampering of important data can be devastating if undetected, compared to outright destruction of the data which is immediately noticed and soon remedied.

6.3.3 Repudiation

Repudiation is when a user plausibly denies performing a specific action — a fancy word for, "It wasn't me!" Non-repudiation is the good security property of having strong evidence so, in the case of a dispute, it can be reasonably proven to a third party exactly who did what (and, ideally, when).

- In our vault example, Evil Attacker denies that they took Rich Customer's money from the vault. They claim that they were at the movies at the time of the crime.
- A user signing a document and then claiming that the action was performed by someone who stole their credentials.
- A user claiming that they never sent an email that they actually sent.
- Denying the contents of a deleted posting online because the screenshot could be easily faked.

6.3.4 Information disclosure

Information disclosure is releasing information to an actor that is not explicitly authorized to have access to that information.

- Evil Attacker quietly snaps a photo of Rich Customer's transaction receipt, and then posts the receipt to Twitter, revealing their account balance.
- An attacker extracting data files from your system (exfiltration).

- Error message revealing too much information, for example, exposing the full path of the program.
- Debug error messages left in the deployed software.
- Using a weak encryption method that is easily cracked, revealing the contents.
- Disclosure of an encryption key, which could disclose the contents of gigabytes of encrypted data that is publicly accessible.

6.3.5 Denial of Service

Denial of service is making a resource, such as a host, server, network or application, unavailable for legitimate users. This category matches loss of availability as described within the C-I-A principles.

- In our vault example, preventing Rich Customer from taking their money from the vault, by putting a fake “bank closed” sign on the door.
- Saturating an online service with a large number of requests (often initiated by orchestrated access from thousands of malware-ridden home computers).
- Taking control of a large number of computers and using them to bombard a server in what is called a distributed denial of service (DDoS, pronounced “dee-doss”) attack.
- Exploiting a buffer overflow and making the system crash.
- Placing an orange cone on the hood of a self-driving vehicle, preventing it from moving.

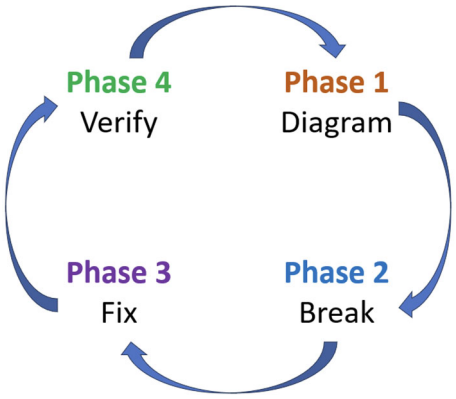
6.3.6 Elevation of privilege

Elevation of privilege is getting access to more resources or functionality than a user is normally allowed by giving them more privileges than intended by the developer or system administrator.

- In our vault example, a bank employee steals the bank president’s ID card so that they can have access to all the bank records.
- Vertical escalation: gaining root or administrator access.
- Horizontal escalation: accessing information associated with a different user.

6.4 The Steps in Microsoft Threat Modeling

Threat Modeling is a key step in the life cycle of any piece of software. Whether you are using Microsoft’s Security Design Lifecycle or another methodology, in the design phase of your project, it is important to understand the potential threats that you must address. In this section, we present a brief overview of the four steps in the Microsoft Threat Modeling process.



6.4.1 Diagram[14]

The goal of this phase is to capture all the requirements and characteristics of your software system. You will need to identify the components and resources of the system, trust boundaries and the way they interact. The end product of this phase is a flow diagram of the system, such as you might draw by hand or using a threat modeling tool.

6.4.2 Break

The second phase is to apply a threat modeling technique to the information that you gathered for the diagram. This phase could be based on a manual approach to threat modeling or an automated tool-based approach such as Microsoft Threat Modeling (Chapter 2.2.1) or PASTA (Chapter 2.2.3).

6.4.3 Fix

For each of the threats that you identified in the second phase, you will need to decide how you will handle that threat. You can respond to a threat in one of three ways:

a. Modify your design so that the threat is no longer applicable. For example, you might discover that you are checking an input constraint outside the trust boundary of your server, so need to do that check inside the boundary (presumably in the host, not in the client).
b. Plan a implementation (coding) strategy that will account for the threat and prevent it from happening. For example, if your design

---

[14] Microsoft actually calls this the “Design” phase. However, this overloads the use of “design”, which is also used for a somewhat different purpose in the SDL steps (as we can see from the diagram at the start of this chapter).

has the threat of a directory traversal attack (Chapter 3.3), then you would need to ensure that you do proper checks on user input before using them to construct a path name.

c. Decide that threat is not a serious concern in your design so ignore it. For example, you might discover that some operation can occur without user authentication. However, it may be that this allowed operation is generally considered low risk and could cause minimal harm.

6.4.4 Verify

The last phase is to review the fixes from the third phase to ensure that the concerns have been addressed and all security controls are in place. As part of this phase, you will identify the information needed to update the flow diagram to include your latest design decisions. You then repeat the process starting from the first phase.

6.5 Microsoft Threat Modeling Tools

There are several tools that help the designer with the non-trivial task of performing Threat Modeling. One of the most mature and widely used is Microsoft's Threat Modeling Tool[15], which (not surprisingly) follows Microsoft's Threat Modeling Methodology.

Here is a list of readings that will introduce you to the tool and how to use it:

- `https://docs.microsoft.com/en-us/azure/security/azure-security-threat-modeling-tool`: (one short page of introduction to MS Threat Modeling).
- `https://docs.microsoft.com/en-us/azure/security/azure-security-threat-modeling-tool-getting-started`: Basic tutorial on the MS Threat Modeling Tool.
- `https://docs.microsoft.com/en-us/azure/security/azure-security-threat-modeling-tool-mitigations`: A description of how to mitigate the threats that you found with the Threat Modeling Tool.

---

[15] J. Geib, B. Santos, D. Coulter, K. Bulloc, J. Howell, M. Baldwin and B. Kess, "Microsoft Threat Modeling Tool", August 2022, `https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool`

6.6 Summary

Microsoft has long been a leader in the area of secure software design. Motivated by security concerns, they developed their Security Design Lifecycle to provide a structured way to produce more secure code and to document this process. As part of SDL, they developed the Threat Modeling methodology, along with a tool, to guide their designers and developers in the secure software process. Drilling down further, they provided the STRIDE system for categorizing threats that a software system might face. Taken together, these provide a good approach to developing secure software.

6.7 Exercises

1. For each of the STRIDE categories, list some applicable threats for a real or theoretical software component. Suggested subjects: a game console, a home assistant (like Alexa or Siri), a smart connected kitchen appliance, a machine learning model like ChatGPT, or something more creative.
2. For a new or existing software project, work through the steps of Microsoft Threat Modeling using the Microsoft tool. Develop the diagram, use the tool to identify potential threats, and then evaluate these to decide what kind of response is needed.
3. Try out the techniques presented in this chapter using the exercise based on Microsoft's Threat Modeling Tool: `https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/Exercises/TM.pdf`