Developer-Driven Threat Modeling

Lessons Learned in the Trenches

This article describes EMC's real-world experiences with threat modeling, including major challenges encountered, lessons learned, and a description of the company's current developer-driven approach.

Danny Dhillon
EMC Corporation

Threat modeling is a conceptual exercise in which we analyze a system's architecture or design to find security flaws and reduce architectural risk (see "Related Work in Threat Modeling"). Threat modeling is most effective at finding architectural security flaws such as failure to authenticate or authorize. Although threat modeling isn't effective for finding specific coding bugs, it can identify classes of coding issues, such as buffer overflow or SQL injection, that could arise due to architectural or technology selection decisions. We can mitigate the flaws identified by modifying the architecture or using the findings of a threat model to better address security throughout the development life cycle by tailoring developer training, proactively implementing coding standards, and focusing code analysis and testing activities on the basis of risk.

Threat modeling can occur at any time during a system's life cycle, but it's recommended during the architecture or design phase, when fixing an issue involves reworking a conceptual design rather than significant reengineering or a patch-release vehicle. Even if not performed early in the architecture or design phase, a threat model can be instrumental in understanding and reducing architectural risk over time. There are a variety of design-risk-analysis and threat-modeling approaches. At EMC Corporation, a technology vendor with a research and development sector of more than 8,000 engineers, we implemented a scalable approach that builds attack knowledge and other security expertise into processes and tools and is effective at finding common security flaws early in the software development life cycle.

A Developer-Driven Threat-Modeling Process

In 2007, EMC began efforts to roll out threat modeling as an integral part of its secure software development processes. The intent was to address security better and embed security considerations into software design processes and throughout the corporation's culture. The threat-modeling process at EMC has evolved over the past few years and currently involves

- creating an annotated dataflow diagram;
- identifying and analyzing threats, guided by a threat library;
- assessing threats' technical risk; and
- mitigating threats to reduce risk.

Because of the distributed product design knowledge and the relative scarcity of security expertise, a threat-modeling approach that software engineers with or without security expertise could leverage was necessary.

Dataflow Diagrams and Annotations

Threat-modeling activities start with a system diagram. We prefer dataflow diagrams because they can visually represent all interaction points that an adversary can leverage to attack a system and also show how data—often the target of attacks—moves through the system.

1540-7993/11/$26.00 © 2011 IEEE COPUBLISHED BY THE IEEE COMPUTER AND RELIABILITY SOCIETIES

Related Work in Threat Modeling

Microsoft coined the term *threat modeling* to describe an attack-focused analysis activity used to find software security flaws. Over the past decade, Microsoft has employed and recommended various threat-modeling methodologies. The first widely published methodology required significant security expertise and understanding of the adversary's view of a system. Over time, Microsoft streamlined the methodology in an effort to make threat modeling more scalable and developer driven.

The most recently published methodology involves a dataflow-diagramming activity followed by a threat-identification session guided by Stride (spoofing, tampering, repudiation, information disclosure, denial of service, and escalation of privilege—an acronym-mnemonic of common software threats). The Stride taxonomy is abstract enough that most attacks on software can be divided into one or more of the categories; however, the methodology requires understanding and applying the abstract Stride threats to a specific piece of software. For example, Stride requires someone with considerable security knowledge to translate spoofing into tangible attacks, such as cross-site request forgery or Web application session hijacking. Without this security knowledge, Stride can't be used effectively.

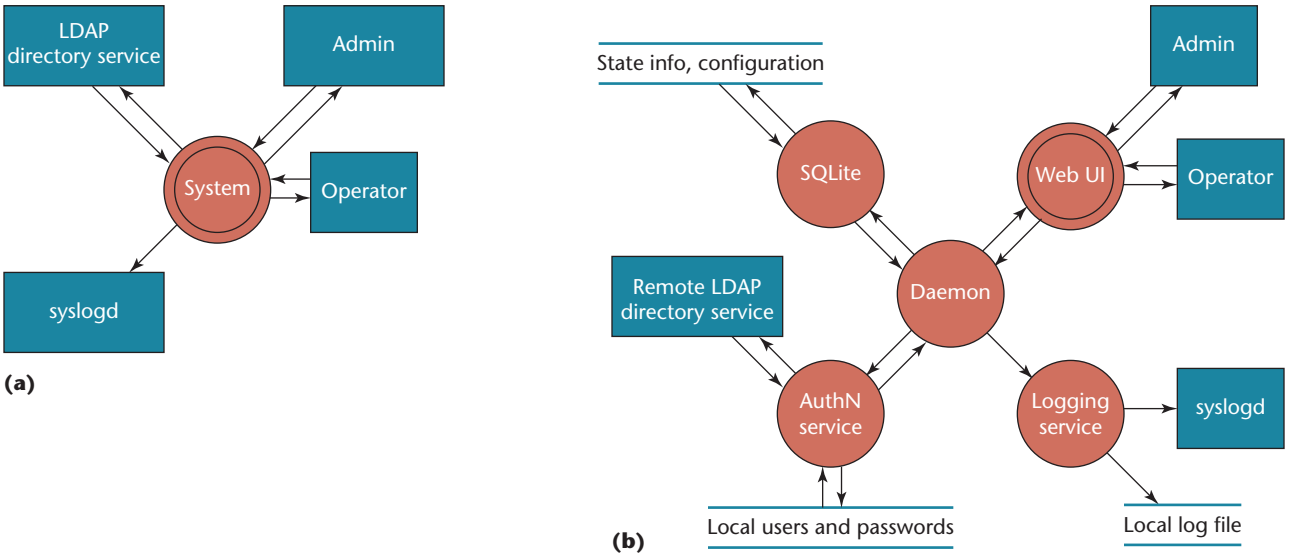

Figure 1. Dataflow diagrams. (a) The context diagram depicts the system as a single element, showing its dataflows and external entities. (b) The level 1 dataflow diagram further decomposes the system, which helps identify additional interaction points that attackers can leverage.

Dataflow diagrams are a simple visual representation of a system. Circles represent processes—the code and components that the development team controls. Double-edged circles are complex processes that are further decomposed in another diagram. Rectangles represent users or systems—often untrusted data sources—that interact with the system being modeled and are outside the development team's control. Double lines represent a place, such as a file or shared memory, where data resides permanently or temporarily. Arrows represent dataflows between other elements (network communications, local communications, or data read/write operations). Each dataflow represents a system entry or exit point, where interactions (or attacks) can occur.

Dataflow diagrams are hierarchical, letting us scope and analyze the system at different decomposition levels. Beyond level 0, the *context diagram*, there are no strict rules on the scope to include; it usually depends on work division in the product organization and the diagrams' readability.

Figure 1a shows an example context dataflow diagram depicting the system as a single element and showing its dataflows with external entities. It scopes the threat-modeling activity but is useful to understand only attacks that exploit external interactions—it doesn't show other types of attacks, including those that exploit flaws in interactions deep in the software that are accessible by external entities. For this reason, software threat models require diagrams that decompose the system into a fair amount of detail.

Figure 1b shows a level 1 dataflow diagram that

further decomposes the same system. Decomposition helps identify additional interaction points that attackers can leverage.

Dataflow diagrams used for structured analysis and design traditionally have used logical elements that don't necessarily map back to concrete software artifacts. In some cases (such as API development), using a logical dataflow diagram might be necessary, but for system-level threat models, we found best results when simple processes are mapped to software artifacts, such as executables, shared libraries, and Java artifacts. We also recommend aggregating similar elements. For example, if a collection of configuration files or log files with similar content are stored in the same location with the same protection mechanisms, we don't draw them separately. With these guidelines in place, dataflow diagrams for EMC systems have had between 10 and 90 elements. We analyzed simpler systems using level 1 diagrams; more complex systems required decomposition of some components to level 2. We haven't needed to decompose to level 3.

Through our experience, we found that dataflow diagram elements alone don't capture all the details necessary to effectively perform a threat model, so we had developers annotate the diagrams to capture additional information. These annotations let threat-modeling efforts more quickly focus on the typical areas in which attackers are interested such as common sources of vulnerabilities and critical protection mechanisms on which the system relies to ensure data trustworthiness. Specifically, we use annotations to note

- processes that run with different privilege levels;
- processes that represent embedded components rather than developed code;
- programming languages used for developed code;
- processes that perform critical security functions, such as authentication, authorization, password management, and cryptographic operations (key generation, encryption, and signing);
- network and local dataflows;
- each dataflow's type—for example, HTTP, LDAP (Lightweight Directory Access Protocol), SQL, API call, command invocation, and file I/O;
- authenticated dataflows;
- encrypted or signed dataflows and data stores;
- data stores that contain sensitive information, such as encryption keys or authentication credentials; and
- how sensitive information is intended to flow through the system.

Figure 2 shows an example of an annotated dataflow diagram. Although we provide guidance on what to annotate, we don't specify how developers should annotate the diagram. Some teams use colors or text on the diagram; others use tables that complement the diagrams.

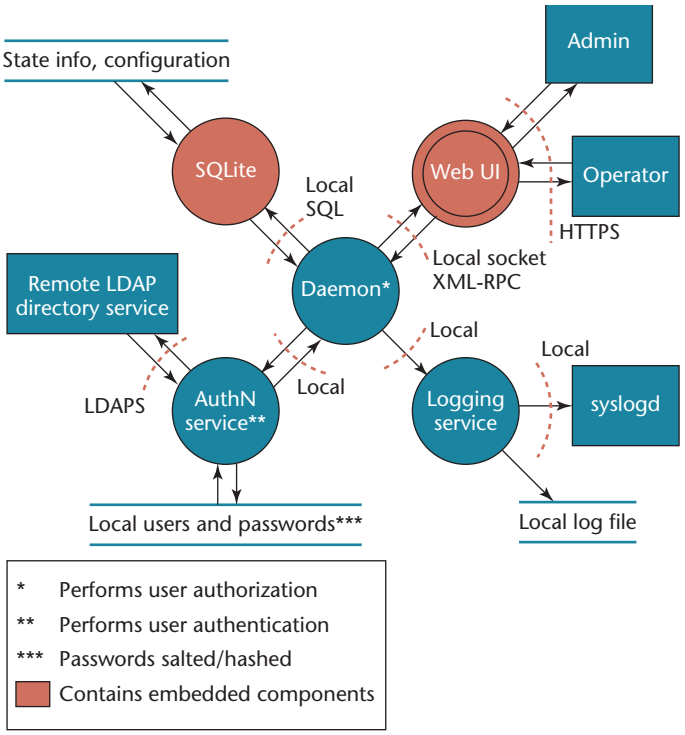

Figure 2. Annotated dataflow diagram. The diagram conveys the elements' security-relevant properties.

Identifying Threats

Threat identification is at the core of threat modeling, yet it's the most difficult step for nonexperts to perform. EMC's initial threat-modeling efforts utilized the Stride (spoofing, tampering, repudiation, information disclosure, denial of service, escalation of privilege) approach to identify threats, but we quickly found that such an approach is inefficient for complex systems and not scalable because it's dependent on attack knowledge. We've streamlined threat modeling and made it less security-knowledge intensive by focusing on interactions and leveraging a "threat library" of attacks.

Analyzing interactions. In our early attempts with Stride, we found analyzing each individual element in the dataflow diagram time-consuming and redundant and began focusing instead on analyzing interactions—the transactions between peer elements. Interactions are a good focal point for threat modeling because a system that can't be interacted with can't be attacked. Focusing on interactions rather than elements yields comparable results with fewer analysis

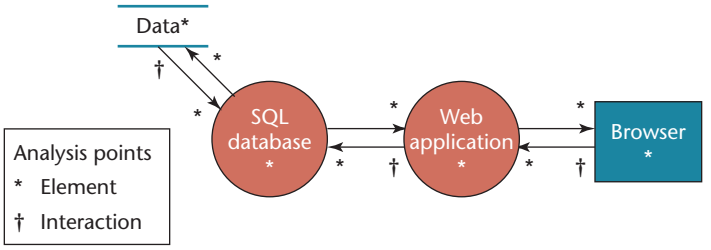

Figure 3. Analyzing interactions versus elements. This diagram has 10 Stride per element analysis points, but only three interaction points. Fewer analysis points results in more efficient analysis.

points. For example, the simple dataflow diagram in Figure 3 has 10 Stride per element analysis points, many of which would lead to redundant threats when analyzed individually. The same diagram only has three interaction points, which each lead to unique threats. Limiting the number of analysis points is important for complex systems especially when threat-modeling resources are limited.

Threat library. Although using interaction points rather than elements streamlined the threat-modeling process, it didn't remove the dependency on security expertise. To lessen this dependency, we created a threat library, which is the cornerstone of EMC's current threat-modeling efforts. The threat library currently contains approximately 35 entries. We initially based them on the Common Weakness Enumeration (CWE) entries that can be identified at design time, but have since reorganized and supplemented them with additional content. The threat library isn't meant to be an all-encompassing list; rather, it captures the minimum considerations we expect a threat-modeling effort to involve and is tailored to the types of systems we develop at EMC. A security team maintains and updates the threat library on the basis of attack developments and experiences at EMC. The major criteria for including entries in the threat library include the following:

- Can we identify the issue at design time?
- Does the issue generally present at least moderate security risk?
- Can we provide actionable mitigation guidance around the issue?

For each threat library entry, we include

- an applicability field that helps developers quickly determine whether the threat is relevant to their system,
- a detailed description of the threat,
- examples that illustrate the threat's implications,
- a baseline risk for the threat with an explanation of how the risk can vary on the basis of the system's details, and
- detailed prescriptive threat mitigation.

We plan to expand the threat library entries to include guidance on how to test mitigations.

Table 1 provides a high-level summary of the threat library entries. It also shows the average Common Vulnerability Scoring System (CVSS v2) score of flaws found from a sample of 30 threat models conducted at EMC. CVSS is an industry standard for assessing the severity of computer system security vulnerabilities.

Assessing Risk

Each threat we identified is prioritized using a CVSS-based risk-ranking tool. The ranking tool we use requires developers to answer 10 yes/no questions about the threat's impact and exploitability. Developers assess only technical risk as part of the threat-modeling process. They don't assess business risk because it's hard to quantify, and developers don't have the knowledge to assess it. Unlike many other risk-assessment models, the ranking tool doesn't require numerical input values, such as probability. We decided to constrain the input values to remove subjectivity from the risk-ranking process and ensure consistent risk scores and definitions across the company. The tool asks the following questions:

- Can the attack result in unauthorized access to private or confidential information?
- Can the attack result in unauthorized changes to the system or external systems?
- Can the attack result in unaudited changes to the system or its data?
- Can the attack result in a permanent denial of service?
- Can the attack be conducted without authentication?
- Does the attack require a highly privileged access role?
- Can the attack be conducted remotely?
- Does the attack require custom exploit code?
- Does the attack require a special condition?
- Does the attack require compromising other components or external systems first?

The first four questions focus on impact and the last six on exploitability. Many of the terms we use, including "private or confidential information," "permanent denial of service," "highly privileged access role," "custom exploit code," "special condition," and "external systems," are further defined in the tool with examples.

The tool is used in conjunction with the threat library, which provides the typical answers for each en-

Table 1. The EMC threat library summary.

Threat	Dataflow diagram pattern	Average Common Vulnerability Scoring System (CVSS) score
Unauthenticated access through user interface	Unauthenticated dataflow	8.2
Access using default credentials	Authenticated dataflow with default or hard-coded credentials	8.9
Attacks on password recovery/change	Process that provides password recovery/change functionality	8.0
Authentication downgrade	Authenticated dataflow	8.9
Attacks on predictable secrets	Process that generates passwords or encryption keys	7.2
Unauthorized access	Incoming dataflow without an authorization check	8.5
Authorization bypass	Process that performs authorization	7.4
Disclosure through interface	Outgoing dataflow to an external entity	7.6
Privilege abuse	Privileged external entity	8.2
Client bypass	Security not enforced in server processes	7.7
Network spoofing	Unauthenticated network dataflow	8.1
Attacks on certificate validation	Certificate-authenticated dataflow	8.9
Network sniffing	Unencrypted network dataflow	7.4
Tampering in transit	Unsigned network dataflow	7.8
Capture/replay attacks	Network dataflow	7.8
Session hijacking	Session-based dataflow	8.8
Attacks on cryptography	Process that performs cryptography	6.5
Online password cracking	Password-authenticated dataflow	8.7
Offline password cracking	Data store that stores passwords	5.8
Tampering through direct access	Unsigned data store	6.5
Disclosure through direct access	Unencrypted data store	5.7
Denial of service through memory/CPU consumption	Process	6.7
Denial of service through storage consumption	Process that writes to a data store	7.2
Denial of service through account lockout	Authenticated dataflow with account lockout mechanism	7.4
Attacks on Web applications (including cross-site scripting, cross-site request forgery, HTTP response splitting, session fixation)	Process that provides Web interface	9.3
Buffer overflows (code injection)	Process written in C or C++	9.3
SQL injection	Outgoing SQL dataflow	9.2
LDAP (Lightweight Directory Access Protocol) injection	Outgoing LDAP dataflow	9.3
XML injection	Outgoing XML dataflow	6.8
OS command injection	OS command invocation dataflow	8.4
Path traversal attacks	Process with dataflow to a file data store	5.5

try and the basis on which the answers might change. The intent is to ensure that risk is calculated consistently and correctly, even if developers aren't completely knowledgeable about the threat's risk ramifications.

The tool translates the questions' yes/no values into numeric values and uses a slightly modified version of CVSS to calculate a numeric risk score. It then generates a qualitative risk ranking of "critical," "high," "medium," or "low" on the basis of this numerical score. We determined the numerical ranges for the qualitative rankings by analyzing past issues at EMC and grouping them into the qualitative buckets. This was preferable to using arbitrary CVSS ranges because it lets us describe the four risk levels in words rather than relying solely on a numerical range definition.

Table 2. Average number of finds per risk level.

Risk	CVSS range	Average number finds
Critical	8.4–10	3.4
High	7–8.39	2.1
Medium	4.5–6.99	1.5
Low	0–4.5	0.1
Total	0–10	7.1

In the future, we might move to a Common Weakness Scoring System (CWSS)-based model. CWSS is an evolving industry standard to prioritize software security errors.

Mitigating Risk

Realizing threat modeling's benefits requires not only identifying issues but also addressing them effectively. The software security field has matured over the past decade, and a wealth of information on how to mitigate many common issues is widely available—but the quality and consistency of that information varies. Moreover, mitigation state of the art and the threat landscape constantly evolve, and developers generally aren't in the best position to separate the good advice from the subpar. For these reasons, our threat library strives to be very prescriptive in terms of recommended mitigation strategies. The mitigation strategies include changes that developers can make during the design phase as well as downstream coding, documentation, and maintenance considerations. Where appropriate, the guidance includes sample code and references to recommended toolkits and frameworks. It also includes alternative mitigations along with their implications and when they should or shouldn't be considered. Alternative mitigations are important not only to support the cases in which there's a valid reason to not use the recommended mitigation strategy but also to educate developers and avoid "reinventing the square wheel."

Results

The current threat-modeling approach has been received favorably and has good traction with EMC development organizations. We've applied the methodology to many storage, resource management, and security software systems that we developed using a variety of programming languages and technologies that run on or embed various operating system and server platforms.

Table 2 summarizes the average number of finds per risk level from 30 recent EMC threat models. Note that these results reflect the fact that our threat library is biased toward high-risk issues to limit the time we spend on threat modeling. If the threat library were comprehensive and also included lower-risk issues, the number of issues and their distribution would change significantly.

We found that developers—who are generally intimate with their system or component design but might not have attack knowledge—achieve better results when guided by a threat library than when guided by Stride, because the threat library lessens dependency on attack knowledge.

The threat library is also a good tool in raising developers' attack knowledge over time. As developers use the threat library, they become educated on common attacks by applying these to their own software designs. This fosters implicit analysis. Over time, even if developers aren't performing a formal threat-modeling activity, they're aware of concrete attacks that they need to mitigate when designing and implementing new features.

Even when security experts are involved, the threat library helps achieve consistent results by providing a baseline of considerations and a framework for dialogue that leads them to focus on the right issues. It helps ensure that common attacks aren't overlooked simply because a particular security expert isn't aware of them.

Moreover, the threat library lets us perform threat modeling with more predictability. When threat modeling is an open-ended brainstorming activity, "When are we done?" is a hard question to answer. When not constrained and not performed by experts, threat modeling can often branch off into far-fetched hypothetical situations that don't have any practical mitigation. This can create a negative perception for everyone involved. We designed our threat library to keep threat-modeling sessions productive—we spend little time on brainstorming and nugatory or redundant analysis. Furthermore, regardless of the dataflow diagrams' complexity, the number of analysis points is determined by the number of threat library entries. Making threat modeling a closed-ended activity that achieves results with predictable effort cost helps get it put on the development schedule.

This approach's obvious disadvantage is that it doesn't consider threats that aren't part of the threat library. Although we encourage product development organizations to go beyond the threat library, developers (as well as many security professionals) generally aren't capable of finding new or creative attacks on their own. A threat-modeling team with appropriate expertise can clearly identify unique threats, but we haven't yet found a way to make this scalable and developer driven.

Even if the threat library isn't capable of finding new attacks, it helps us more efficiently utilize our security expertise. Product development organizations armed with the threat library can create a threat model for standard types of software, freeing security experts to spend more time modeling software likely to contain new threat types. These results continuously feed into our threat library.

We're currently implementing tools that will help us consistently capture annotated dataflow diagrams of our software designs and automate the threat-identification process. Given an annotated dataflow diagram, the tool will use a set of analysis rules to generate a list of potential threats that the development team can review to confirm applicability and plan mitigation.

We're also looking at ways to tie threat modeling with guided security testing. This will let us better validate the threat models that we create and ensure that they represent the final developed systems. ☐

Danny Dhillon *is a principal security engineer for EMC Corporation's Product Security Office. His interests include secure design, threat modeling, security for virtualized and converged datacenters, routing security for mobile ad hoc networks, and threshold cryptography. Dhillon has an MA in liberal arts from Harvard University. Contact him at danny.dhillon@rsa.com.*