

Defining Misuse within the Development Process

The software development industry often brings in security at the eleventh hour, right before developers throw the code over the wall—that is, deploy it into production—and ask, “Well, is it secure?” At this point, hilarity—for the objective observers, anyhow—

ensues as security personnel work feverishly to shove crypto, firewalls, and all the other mechanisms at their disposal into the most egregious risk areas.

To combat this antipattern, the software security discipline has worked to instantiate itself closer to the beginning of the software development life cycle (SDLC). After a software project is signed off, use cases represent the earliest opportunity for involvement. Misuse cases, a concept covered in the May/June 2005 issue,¹ prescribe one such way for security to involve itself in early brainstorming. That article outlined misuse cases as a way to help analysts characterize what misuses or abuses attackers could promulgate against a system. This article extends this outline to how to create useful misuse cases within the development process.

Use vs. misuse cases

Use cases describe what behaviors a system exhibits on behalf of its stakeholders from an actor’s viewpoint. The use case format explicitly conveys each actor’s goals and the flows the system must implement to accomplish them—for example, use cases for a hotel booking system will include actors such as a travel agent, a customer, a desk clerk, and an auditor, so each actor will have very dif-

ferent goals and experiences. The level of abstraction a use case model offers makes it an appropriate starting point for software security analysis and design activities.

In contrast, a misuse case describes potential system behaviors that a system’s stakeholders deem unacceptable. In a misuse case, at least one threat (or, in more common parlance, attacker) serves as an actor. Thus, a misuse case conveys each threat actor’s goals in misusing the system. It’s important that these misuses either represent high-probability attacks or high-impact events that negatively affect the system’s legitimate stakeholders. Misuse cases should be at a level of detail that drives design activities. By considering conceptual attacks, such as types of theft, privacy violation, and denial of service, the misuse case prevents modeling analysts from becoming stymied or inappropriately mired in unimportant (at the time) technical details. Like use case models, misuse cases are iteratively refined throughout the software development life cycle. Table 1 describes the key elements in use and misuse case models for that stage of development.

Like use cases, misuse cases benefit from focusing on the actor’s perspective—for example, in systems that divide perimeters into zones,

misuse cases show which attacks threat actors can feasibly execute from each zone. By localizing threats, the security architect finds traction points to move the security architecture and design process forward.

What to model

Because modelers create misuse cases very early in the SDLC, they must be comfortable with many unknowns. Specifically, modelers shouldn’t expect to find full access to data schemas, method interfaces, and so forth. Key technologies might not even be selected yet. Don’t let this ambiguity cloud the modeling effort—instead, focus on what conceptual attacks the system must be resilient against. Misuse cases aren’t an end goal—rather, they’re an incremental step to help focus software security efforts going forward; misuse cases should expect to iteratively improve time.

Let’s look at a basic example. In Figure 1, an authenticated and authorized bank customer uses a Web interface to review his or her accounts and then transfer funds. The transfer money use case traverses two system boundaries and updates the bank’s record system. As we consider this example, two questions immediately come to mind: Will the transfer money use case synchronously or asynchronously perform the updates? What application servers run in the bank’s DMZ? Many key decisions are likely to be in flux or subject to change, but at this stage of software development, we don’t document misuse cases to audit the system’s security. We save these deeper technical questions and their answers for design activities.

GUNNAR
PETERSEN
Arctec Group

JOHN STEVEN
Cigital

Table 1. Core elements in use and misuse case models.

| ELEMENTS | USE CASE MODEL | MISUSE CASE MODEL |
|----------------|---|---|
| Actor | Set of authorized systems and users interacting with the system. | Threats including internal or external attackers and competitors using the system in unexpected ways. A threat can also be an authorized system user. |
| Goal | Stakeholders' business objective accomplished by actors in a particular workflow. | Threat's objective to affect a system's stakeholders (such as gaining access to a system or data). |
| Use case | Set of desired features and behaviors, expressed as workflows, that a system must perform on each actor's behalf. | Set of vulnerability exploitation paths (such as attack vectors that the system should resist). |
| Preconditions | Required system state at the beginning of use case execution. | Required conditions for a successful exploit (such as the actor must have gained access to the corporate intranet). |
| Postconditions | System's state at the end of execution. | Possible outcome if attacker succeeds (such as system is unavailable to authorized users). |

The example in Figure 2 layers three different actors and misuse cases onto the scenario in Figure 1. Here, a unique threat misuses the system, using the Internet to inject commands and exploit the transfer money use case to gain deeper access into the system. The diagram shows the DMZ threat intercepting the update message before it reaches the record system, whereas the bank threat alters the updates in the record system itself.

Note that each threat requires a set of preconditions in terms of access to various systems. In some circumstances, threats must compose attacks to misuse the system with an appreciable impact. In these cases, attacks serve as additional preconditions for more complex misuse cases. Likely candidates for precondition attacks include delivery attacks, social engineering, or interposition attacks.

Misuse vs. error conditions and alternative flows

Misuse cases can stem from alternative flows and error conditions within normal use cases. Indeed, when existing use cases thoroughly document error conditions, they serve as good muses for creating misuse cases. A “log in” use case, for example, might document the

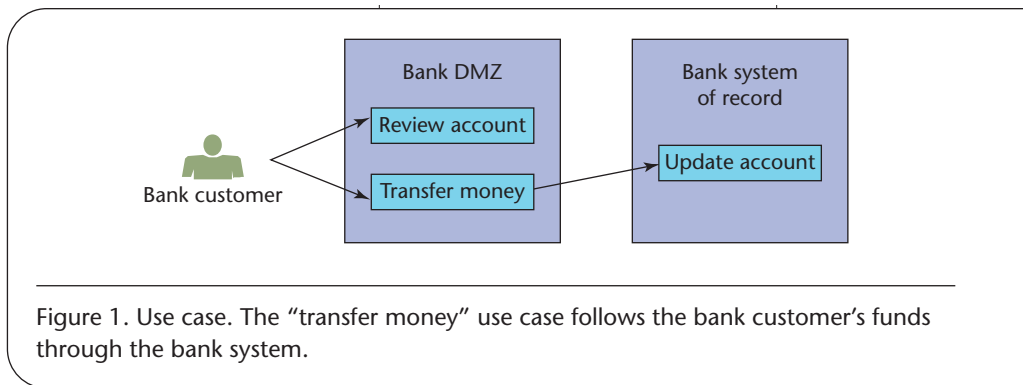


Figure 1. Use case. The “transfer money” use case follows the bank customer’s funds through the bank system.

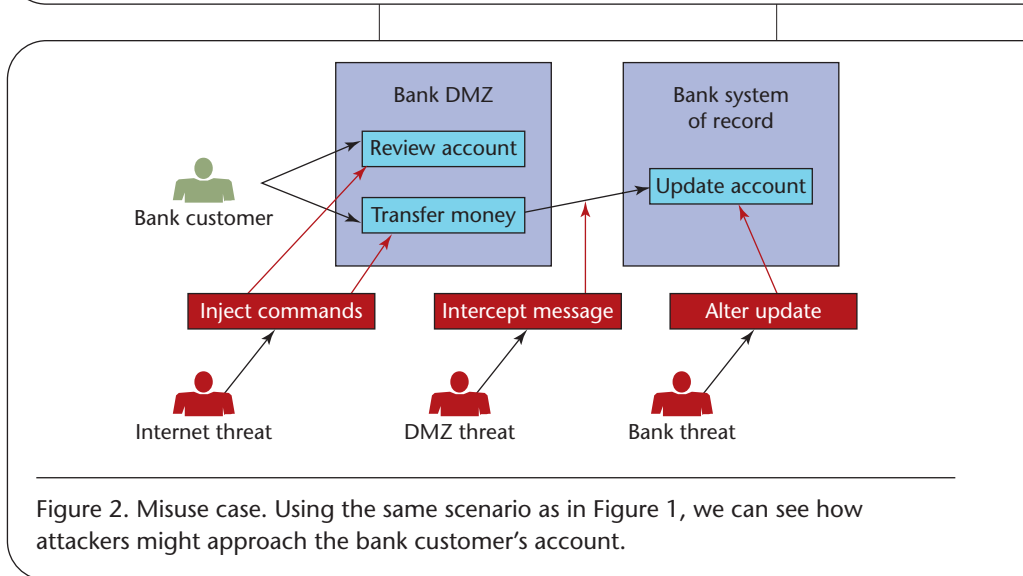


Figure 2. Misuse case. Using the same scenario as in Figure 1, we can see how attackers might approach the bank customer’s account.

scenario in which an actor enters a username with too many characters—an error condition. A misuse case, representing the workflow driven by an unauthorized threat, might take misuse one step further

by specifying a type of phishing attack in which the threat actor enters a username, SQL command delimiter, and another SQL command (such as `bob || SELECT * FROM USERS`) into the software’s authenti-

Misuse case tools

The most important factor in misuse case modeling is to use tools that integrate with what the development team uses to write use cases. You don't need fancy Unified Modeling Language tools to get started. Thoughtworks' chief scientist Martin Fowler states that his regular modeling tool is Visio (www.martinfowler.com/bliki/UmlSketchingTools.html). Jaczone offers a free use case modeling tool called Essential Modeler (<http://jaczone.com/product/tryit/#essentialmodeler>). A basic diagramming tool, word processor, and creative spirit are the only prerequisites to getting started.

cation form. Rather than an honest error ("Oh, I didn't know that I'm only allowed a 10-character username"), the actor proactively tries to subvert the system ("I don't have a valid login, but I'd like a list of all valid usernames.").

Misuse cases address several key issues in most organizations' SDLCs, by

- identifying a system's threats and their probable interactions at a conceptual level, with the software's interfaces;
- narrowing discussions by requirement analysts, architects, and developers in high-impact or probable "what" and "where" attack scenarios;
- notifying architects of stakeholders' specific fears through synthetic analysis for modeling security architecture and further iterations;
- anchoring security guidance in all system workflows ("The J2EE authorization model doesn't help mediate access to data through the database's reporting interface."); and
- giving timely design feedback early in the SDLC process and providing concrete scenarios to address with the software security architecture, design, and test plans.

These actions get security out of seagull mode (swooping into a meeting, making a lot of noise, and flying away) and help security function as an equal design partner in the SDLC. Threat models provide similar utility as more detailed design and development phases continue.

So where do misuse cases end and

threat models begin? Misuse cases describe the "what" and "where" from the attacker's side, and threat models provide the "how." Misuse cases don't typically provide much technical detail, but threat models do, describing detailed attack paths, interfaces, and data elements that a threat exploits. Think of misuse cases and threat models working together in the same way that use cases provide the structure and context for detailed Unified Modeling Language design documents, such as class and sequence diagrams.

Preparing a misuse case

Use cases can't be created without understanding stakeholders' goals. Likewise, misuse cases can't be created without understanding what eventualities negatively impact those goals. Rather than starting a heavy interrogation of system stakeholders, those responsible for misuse case creation should enumerate each stakeholder's doomsday scenario. Each stakeholder can easily create a list of one to three circumstances in which system malfunction would create a "game over" situation of inarguable consequence.

Resources, especially Bruce Schneier's *Secrets and Lies*,² help modelers understand how to express a system's security goals. In turn, by understanding each goal's subtle differences (such as the difference between offering user confidentiality and privacy or anonymity), modelers can come up with several misuses. Understanding principles of secure design provides value, but modelers

should stop short of incorporating attack patterns into misuse cases. Keep misuse cases at the level of threats. Conversation about attacks on technologies or attack patterns likely represents a rabbit hole and should be avoided at this stage of the SDLC. Also, keep *Exploiting Software*³ and *19 Deadly Sins*⁴ on your shelf until you use misuse cases to conduct destructive security test planning or architectural analysis. You won't need them at this stage, either.

So where do misuse cases end and threat models begin? Misuse cases describe "what" and "where" from the attacker's viewpoint, and threat models provide the "how." Thus, misuse cases show attackers and their goals, but don't typically provide technical details. Threat models augment misuse cases by describing the detailed attack paths, interfaces, and data elements these threats will exploit. Misuse cases and threat models work together in a way analogous to use cases and UML design documents such as class and sequence diagrams.

Architectural considerations

The Internet threat's ability to inject commands in Figure 2 warrants the need for further input validation mechanisms. The DMZ threat's intercept message misuse case requires message-level security mechanisms such as digital signatures and guaranteed delivery services such as WS-ReliableMessaging and Java Message Service, the bank threat's alter update misuse case might drive the need for additional database security, audit logging, and database access control protections. This simple example shows that different points in the system require context-specific protections whereas a dualistic security model that proposes controls to let system as a whole be judged as "secure" or "not secure" (a common alternative to the risk-based approach of constructing misuse cases) misses the mark. Architects and developers must know about these security con-

trols with some degree of specificity and where they live in the overall system.

Producing misuse cases means iteratively poring over use cases and folding in stakeholders' negative impacts. Consuming misuse cases means iterating over them and folding in specification or design for resisting the attacks they describe or imply. This is not about perfect security: it's about security developers rolling up their sleeves and functioning as a design partner in the SDLC. Sounds simple, no? If the organization uses Agile or XP-style user stories or other synthetic requirements artifact instead of use cases, the security team should also work to map security concerns to those formats.

Misuse cases provide a discrete set of architectural concerns from a software security architecture viewpoint. They also provide a framework for more detailed threat modeling and architectural risk analysis, and can provide focus for white-box penetration efforts and source code analysis efforts. In your own organization, problem and solution patterns will likely emerge over time. As your organization identifies these patterns, remember that they can help define security patterns to deal with attacks that occur across many systems, eventually evolving your enterprise to a more mature security posture driven by solutions that work in known contexts. □

References

1. P. Hope, G. McGraw, and A.I. Antón, "Misuse and Abuse Cases: Getting Past the Positive," *IEEE Security & Privacy*, vol. 2, no. 3, 2004, pp. 90–92.
2. B. Schneier, *Secrets and Lies: Digital Security in a Networked World*, John Wiley & Sons, 2004.
3. G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*, Addison-Wesley, 2004.
4. M. Howard, D. LeBlanc, and J. Viega, *19 Deadly Sins of Software Security*, McGraw-Hill, 2005.

Gunnar Peterson is a founder and managing principal at Arctec Group, which supports clients in strategic technology decision making and architecture. His work focuses on distributed systems security architecture, design, process, and delivery. Contact him at gunnar@arctecgroup.net.

John Steven is a technical director and software security principal at Cigital. His interests include J2EE security, and he works in partnership with companies large and small to help them build their own software security capabilities internally. Steven has an MS in computer science and a BS in computer engineering from Case Western University. Contact him at jsteven@cigital.com.