Misuse and Abuse Cases: Getting Past the Positive

Paco Hope and Gary McGraw
Cigital

Annie I. Antón
North Carolina State University

Software development is all about making software do something: when software vendors sell their products, they talk about what the products do to make customers' lives easier, such as encapsulating business processes or something similarly positive. Following this trend, most systems for designing software also tend to describe positive features.

Savvy software practitioners are beginning to think beyond features, touching on emergent properties of software systems such as reliability, security, and performance. This is mostly because experienced customers are beginning to demand secure and reliable software; but in many situations, it's still up to the software developer to define "secure" and "reliable."

To create secure and reliable software, we first must anticipate abnormal behavior. We don't normally describe non-normative behavior in use cases, nor do we describe it with UML, but we must have some way to talk about and prepare for it. "Misuse" (or "abuse") cases can help organizations begin to see their software in the same light that attackers do. By thinking beyond normative features, while simultaneously contemplating negative or unexpected events, software security professionals can better understand how to create secure and reliable software.

Guttorm Sindre and Andreas Opdahl extend use-case diagrams with misuse cases to represent the actions that systems should prevent in tandem with those that they should support for security and privacy requirement analysis.[1] Ian Alexander advocates using misuse and use cases together to conduct threat and hazard analysis during requirements analysis.[2] In this article, we provide a nonacademic introduction to the software security best practice of misuse and abuse cases, showing you how to put the basic science to work. In case you're keeping track, Figure 1 shows you where we are in our series of articles about software security's place in the software development life cycle.

Security is not a set of features

There is no convenient security pull-down menu that will let you select "security" and then sit back and watch magic things happen. Unfortunately, many software developers simply link functional security features and mechanisms somewhere into their software, mistakenly assuming that doing so addresses security needs throughout the system. Too often, product literature makes broad, feature-based claims about security, such as "built with SSL" or "128-bit encryption included," which represent the vendor's entire approach for securing its product.

Security is an emergent property of a system, not a feature. This is like how "being dry" is an emergent property of being inside a tent in the rain. The tent only keeps you dry if the poles are stabilized, vertical, and able to support the weight of wet fabric; the tent also must have waterproof fabric (with no holes) and be large enough to protect everyone who wants to remain dry. Lastly, everyone must remain under the tent the entire time it's raining. So, although having poles and fabric is important, it's not enough to say, "the tent has poles and fabric, thus it keeps you dry!" This sort of claim, however, is analogous to the claims that software vendors make when they highlight numbers of bits in crypto keys or the use of particular encryption algorithms. Cryptography of one kind or another is usually necessary to create a secure system, but security features alone are not sufficient for building secure software.

Because security is not a feature, it can't be bolted on after other software features are codified, nor can it be patched in after attacks have occurred in the field. Instead, it must be built in from the ground up, as a critical part of the design from the very beginning (requirements specification) and included in every subsequent development phase all the way through fielding a complete system.

Sometimes building security in at the beginning means making explicit trade-offs when specifying system requirements. For example, ease of use might be paramount in a medical system designed for secretaries in doctors' offices, but complex authentication procedures, such as obtaining and using a cryptographic identity, can be hard to use.[3] Furthermore, regulatory pressures from HIPAA and California's new privacy regulations (SB 1386) force designers to negotiate a reasonable trade-off.

Technical approaches must go far beyond the obvious features, deep into the many-tiered heart of a software system to provide enough security: authentication and authorization can't stop at a program's front door. The best, most cost-effective approach to software security incorporates thinking beyond normative features and incorporates that thinking throughout the development process. Every time a new requirement, feature, or use case is created, someone should spend some time thinking about how that feature might be unintentionally misused or intentionally abused. Professionals who know how features are attacked and how to protect software should play an active role in this kind of analysis.

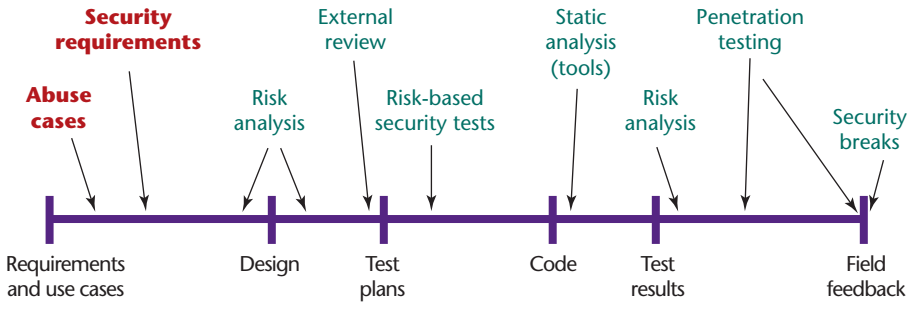

Figure 1. The software development life cycle. Throughout this series, we'll focus on specific parts of the cycle; here, we're examining abuse cases and security requirements.

Thinking about what you can't do

Attackers are not standard-issue customers. They're bad people with malicious intentions who want your software to act to their benefit. If the development process doesn't address unexpected or abnormal behavior, then an attacker usually has plenty of raw material with which to work.[4]

Attackers are creative, but despite this, they'll always probe well-known locations—boundary conditions, edges, intersystem communication, and system assumptions—in the course of their attacks. Clever attackers will try to undermine the assumptions on which a system was built. If a design assumes that connections from the Web server to the database server are always valid, for example, an attacker will try to make the Web server send inappropriate requests to access valuable data. If the software design assumes that the client never modifies its Web browser cookies before they are sent back to the requesting server (in an attempt to preserve some state), attackers will intentionally cause problems by modifying the cookies. *Building Secure Software* teaches us that we have to be on guard with all of our assumptions.[5]

When we design and analyze a system, we're in a great position to know our systems better than potential attackers do. We must leverage this knowledge to the benefit of security and reliability, which we can do by asking and answering the following critical questions: What assumptions are implicit in our system? What kinds of things make our assumptions false? What kinds of attack patterns will an attacker bring to bear?

Unfortunately, a system's creators are not the best security analysts of that system. Consciously noting and considering all assumptions (especially in light of thinking like an attacker) is extremely difficult for those who have built up a set of implicit assumptions. Fortunately, these professionals make excellent subject matter experts (SMEs). Together, SMEs and security analysts can ferret out base assumptions in a system under analysis and think through the ways an attacker will approach the software.

Creating useful misuse cases

One of the goals of misuse cases is to decide and document a priori how software should react to illegitimate use. The simplest, most practical method for creating misuse cases is usually through a process of informed brainstorming. Several theoretical methods require fully specifying a system with rigorous formal models and logics, but such activities are extremely time and resource intensive. A more practical approach teams security and reliability experts with SMEs. This approach relies heavily on expertise and covers a lot of ground quickly

To guide brainstorming, software security experts ask many questions of a system's designers to help identify the places where the system is likely to have weaknesses. This activity mirrors the way attackers think. Such brainstorming involves a careful look at all user interfaces (including environmental factors) and considers events that developers assume a person can't or won't do. These "can'ts" and "won'ts" take many forms: "Users can't enter more than 50 characters because the JavaScript code won't let them, " or "Users don't understand the format of the cached data. They can't modify it." Attackers, unfortunately, can make these can'ts and won'ts happen.

The process of specifying abuse cases makes a designer very clearly differentiate appropriate use from inappropriate use, but to get there, the designer must ask the right questions: How can the system distinguish between good and bad input? Can it tell whether a request is coming from a legitimate or a rogue application replaying traffic? All systems have more vulnerable places than the obvious front doors, of course, so where can a bad guy be positioned? On the wire? At a workstation? In the back office? Any communication line between two endpoints or two components is a

Attack patterns

Attack patterns are extremely useful in generating valid abuse and misuse cases. The book *Exploiting Software* (Addison-Wesley, 2004) includes the identification and description of 48 attack patterns, 10 of which are shown here:

Make the client invisible
Target programs that write to privileged OS resources
Attack user-supplied configuration files that run commands to elevate privilege
Leverage configuration file search paths
Manipulate terminal devices
Perform simple script injection
Pass local file names to functions that expect a URL
Inject functions into content-based file systems
Use alternative IP addresses and encodings
Force a buffer overflow in an API call

place where an attacker can try to interpose, so what can this attacker do in the system? Watch communications traffic? Modify and replay such traffic? Read files stored on the workstation? Change registry keys or configuration files? Be the DLL? Be the "chip"?

Trying to answer such questions helps software designers explicitly question design and architecture assumptions, and it puts the designer squarely ahead of the attacker by identifying and fixing a problem before it's even created.

An abuse case example

Cigital recently reviewed a client-server application and found a classic software security problem. The architecture had been set up so that the server relied on the client-side application, which manipulated a financially sensitive database, to manage all data-access permissions—no permissions were enforced on the server itself. In fact, only the client had any notion of permissions and access control. To make matters worse, a complete copy of the database (only parts of which were to be viewed by a given user with a particular client) was sent to the client program, which ran on a garden-variety desktop PC. This means that a complete copy of the sensitive data expressly *not* to be viewed by the user was available on that user's PC in the clear. If the user looked in the application's cache on the hard disk and used a standard-issue unzip utility, he or she could see all sorts of sensitive information.

The client also enforced which messages were sent to the server, honoring these messages independent of the user's actual credentials. The server assumed that any messages coming from the client had passed the client software's access control system (and policy) and were, therefore, legitimate. By intercepting network traffic, corrupting values in the client software's cache, or building a hostile client, malicious users could inject data into the database that they were not even supposed to read (much less write to).

Determining the can'ts and won'ts in such a case is difficult for those who think only about positive features. Attack patterns can provide some guidance (see the sidebar). Attack patterns are like patterns in sewing—a blueprint for creating an attack. Everyone's favorite example, the buffer overflow, follows several different standard patterns, but patterns allow for a fair amount of variation on a theme. They can take into account many dimensions, including timing, resources required, techniques, and so forth.[4] When we're trying to develop misuse and abuse cases, attack patterns can help.

Of course, like all good things, misuse cases can be overused (and generated forever with little impact on actual security). A solid approach to building them requires a combination of security know-how and subject matter expertise to prioritize misuse cases as they are generated and to strike the right balance between cost and value. □

References

1. G. Sindre and A.L. Opdahl, "Eliciting Security Requirements by Misuse Cases," *Proc. 37th Int'l Conf. Technology of Object-Oriented Languages and Systems* (TOOLS-37'00), IEEE Press, 2000, pp. 120 –131.
2. I. Alexander, "Misuse Cases: Use Cases with Hostile Intent," *IEEE Software,* vol. 20, no. 1, 2003, pp. 58–66.
3. A. Whitten and J. Tygar, "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0," *Proc. Usenix Security Symp.*, Usenix Assoc., 1999.
4. G. Hoglund and G. McGraw, *Exploiting Software*, Addison-Wesley, 2004.
5. J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, 2001.

Paco Hope *is a senior security consultant with Cigital. His areas of expertise include application security, cryptography, LAN and host security, and smart cards. He is co-author of* Mastering FreeBSD and OpenBSD Security *(O'Reilly, 2004). He has an MS in computer science from the University of Virginia. Contact him at paco@cigital.com.*

Annie I. Antón *is an associate professor in the North Carolina State University College of Engineering, where she is a Cyber Defense Lab member and director of The Privacy Place (theprivacyplace.org). Her research interests include software requirements engineering, Internet privacy and security policy, software evolution, and process improvement. She has a BS, MS, and PhD in computer science from the Georgia Institute of Technology. She is a member of the ACM, the IAPP, and a senior member of the IEEE. Contact her at aianton@eos.ncsu.edu.*

Gary McGraw *is chief technology officer of Cigital. His real-world experience is grounded in years of consulting with major corporations and software producers. He serves on the technical advisory boards of Authentica, Counterpane, Fortify, and Indigo. He also is coauthor of* Exploiting Software *(Addison-Wesley, 2004),* Building Secure Software *(Addison-Wesley, 2001),* Java Security *(John Wiley & Sons, 1996), and four other books. He has a BA in philosophy from the University of Virginia and a dual PhD in computer science and cognitive science from Indiana University. Contact him at gem@cigital.com.*