ENTERPRISE SOFTWARE SECURITY

A CONFLUENCE OF DISCIPLINES

Kenneth R. van Wyk ▪ Mark G. Graff
Dan S. Peters ▪ Diana L. Burley, Ph.D.

Enterprise Software Security

kitchen" sort of situation. It is one thing to reach out to a stakeholder and solicit input; that is completely different than handing the helm over to the stakeholder.

In our experience, we've often found design review processes in which various stakeholders are included, but more often than not, the gating functions have been primarily business related and not security related. For example, can the application be developed within budget? Will it be delivered on time? Is the expense justifiable? Although this is all good and essential to do, we feel that security concerns need to also be included as early as possible, and that means including the security stakeholders in the process.

Requirements

Although many software developers these days eschew the practice of formally gathering and documenting their software requirements, there are many things worth considering at this earliest stage of development. Even if this is done only at an informal or "whiteboard" level, it can significantly help the team in understanding and capturing a project's security needs in addition to its functional needs.

We'll describe these considerations and steps here in several areas: abuse cases, regulatory requirements, and security requirements. Later, we will consider these requirements together with the security tiers we described earlier in the chapter. All these will come together as we discuss the topic of secure designs later in this chapter. Although the overall process is described as a team exercise, the role of the SA is extremely important throughout these activities, because he or she serves as both an anchor and a guiding force for all the participants.

Abuse Case Analyses

To start with, although abuse case analyses had been used in various ways for some time, McGraw's *Software Security: Building Security In*[1] provides us with one useful description of abuse case analysis. In essence, abuse case analysis looks at the intended functionality of a piece of software and seeks ways in which the software can be misused for evil purposes. As such, it is a review-based process to help us ensure that we're not building something that can be used to cause harm. That said, abuse case analysis can be a powerful means of finding problems with a project before it ever begins.

If the software will likely be misused or abused in a way the owner really does not want to happen, it is a serious problem.

Let's illustrate this with an example. Suppose you're the engineering team leader of your company's customer-facing web presence. One day, the vice president of marketing walks into your office and asks you to add a new feature to the web application: a mechanism for customers to subscribe to a new monthly newsletter the marketing department is launching. Simple enough; you can add a basic web form that asks the customers for their email address and perhaps some other contact data. After you have the information, you simply add the incoming addresses into a database of customers who receive the newsletter. All done? What could go wrong with this scenario? After all, all the functionality that the VP asked for is now complete, right?

Although it's true that this scenario fulfills all the functional requirements, there's a big problem. You probably recognized immediately that anyone could enter a "customer's" information and have her added to the subscription list. That's an abuse case. Heck, someone who really wanted to disrupt us could write a short script that would submit thousands or millions of addresses into our database if we're not careful. That's another abuse case, and one with obvious and really bad consequences. Now, let's take that further, to its logical conclusion.

If we recognize the potential for abuse, we'd want to prevent that from happening, naturally. A first step could be to add a security requirement to the functional requirement that might say something like "only verified email addresses may be added to the subscriber list." It's a good, actionable requirement. Our development team might implement that by sending an email confirmation to each address submitted for inclusion in the subscriber list. Now are we done?

Not so fast. Let's think a bit fiendishly here. If an email confirmation goes to the (intended) subscribers and requires them to verify that they want to be on the list, what could go wrong? Well, there's still an abuse case potential here. The mere act of sending out those confirmation emails could be disruptive. If an attacker bombards our subscription mechanism with fake but carefully chosen email addresses—say, at one of our key business partners—what would happen if our system then sends thousands and thousands of confirmation emails?

So it's not enough to send out a confirmation email; we have to ensure that our application is talking to a human, and not a script. There's another security requirement to consider. We note that CAPTCHAs are routinely used to address this issue. (CAPTCHAs are automated tests used to verify that a user is in fact a human. They usually show a distorted image of a word or phrase that an artificial intelligence would be unable to recognize but the user can read and enter correctly.) Nonetheless, let's add a security requirement such as "subscription requests may be issued only by human users of the system." See where this is going?

It's always best to consider abuses such as the ones we've described here before a system is rolled out into a production environment. But that requires the development team to be able to really think fiendishly, ignoring the mere functional requirements, and to consider how the system can be abused. It has been our experience that this can be a difficult leap for many developers. Security professionals, on the other hand, have been worrying about abuses like this for decades, and thinking fiendishly comes naturally to them. Invite them to participate.

In considering abuse cases, the following are some questions and important areas of concern to consider for each application. These questions are similar to those we'll address while doing a threat model, but let's consider them separately here while we ponder abuse cases.

➤ **How?—Means and Capabilities**

- Automated versus manual

In our mailing list scenario given previously, we saw an automated attack against a simple function. Often, designers consider a single use case with blinders on when thinking about how an application might be used. In doing this, they fail to see how the (usually simple) act of automating the functionality can be used to wreak significant havoc on a system, either by simply overwhelming it or by inserting a mountain of garbage data into the application's front end. Never underestimate the determination of a `while true do {}` block.

➤ **Why?—Motivations and Goals**

- Insider trading

Automating a user interface into an application is in no way the end of the myriad of ways an attacker can abuse an application.

Consider the human aspects of what an application will be capable of doing, and what sorts of bad things a maliciously minded person might be able to make of those capabilities. Insider trading should be a significant concern, particularly in publicly traded companies. Automation is, after all, a double-edged sword of sorts. We not only are automating a business function, but also might well be inadvertently automating a means for someone to attack a business function.

- **Personal gain**

Similarly, look for avenues of personal gain in an application. Ask whether a user of the application could use the information to "play" the stock market, for example, in a publicly traded company. This can be a significant concern in major business applications in enterprise environments.

- **Information harvesting**

Here, we look for opportunities for an authorized application user to gather—perhaps very slowly over time—information from an application and use that information for bad purposes. A prime example could include a customer database that contains information on celebrity or otherwise VIP customers, such as a patient database in a hospital where the VIP has been treated. That information could be very valuable on the black market or if sold to the media.

- **Espionage**

Although several of these issues overlap significantly, it's useful to consider them separately. Espionage, whether corporate or otherwise, could well be simply a case of information harvesting, but it's still worthy of separate consideration. Consider not just information like the celebrity database, but also company proprietary information and how it could be collected and sold/given to a competitor. What opportunities does the application being analyzed offer up to a user who might be persuaded to do such a thing?

- **Sabotage**

  Even in the best of economic climates, you'll occasionally find disgruntled employees who are bent on damaging a company for all manner of reasons. Their actions might be clear and unambiguously malicious—such as deleting files or destroying records in a company database—but they might also be more subtle and difficult to detect. Consider how a malicious-minded application user might be able to harm the company by sabotaging components in an application.

- **Theft**

  This one is sort of a catchall for things that weren't brought up in the previous ones, but it's worthwhile considering general theft at this point. Credit card account information is a prime candidate here.

Now, it's quite likely a software developer will look at a list like this and throw her arms up in the air in frustration, thinking it's not feasible to brainstorm something like this comprehensively. After all, it is fundamentally an example of negative validation, which we generally seek to avoid at all costs. Although that's true, there's still significant merit in doing abuse case analysis. Of course, the secret to getting it right is to do it collaboratively with some folks who are practiced at this sort of thing—like, say, the information security team.

It is also a good idea to consider separately the likelihood of an attack and the impact of a successful attack. These two things are quite different and bear separate analysis. Impacts can be imagined or brainstormed quite effectively, whereas likelihood can be more deeply analyzed, or even quantified.

Here's how the collaborative approach can work for analyzing abuse cases. After you've gathered a basic understanding of the functional goals of your project, invite a few key folks to take a look at the project and "throw stones" at it. You will want to ensure that all the interested parties are at the meeting; these should include at a minimum the business process owner, the design team, the information security team and/or incident response team, and the regulatory compliance monitoring team.

Tips on Conducting a Successful Abuse Case Study

To set the stage for considering abuse cases, consider holding a meeting among the key stakeholders: business owner (representative), IT security and/or CSIRT, security architecture, and developers. Before the meeting, be sure to distribute some basic design information about the application being considered. (But still bring copies of these documents to the meeting, just in case anyone "forgot" his copy.) This can be fairly high level at this point, but it must include a list of the business requirements and a graphic visualization of how the application should function, along with some basic narrative descriptions of the application's components and what they do. Data flow diagrams can be useful here as well.

At the meeting, carefully introduce the purpose of the meeting to all the participants. Emphasize that you are only looking for misuse or abuse cases at this time. Your participants will invariably head down the SQLi or XSS path, but steer them back and focus exclusively on how the application's core functionality could be abused.

To catalyze constructive discussion, orally describe the application's functions through its phases: startup, production (including transactions, queries, or whatever else the application does), and how it shuts down. Next describe the normal use cases of each class of application user, emphasizing the types of information each user can access and what functionality she is presented with in the application.

You should be seeking areas of debate at this point. Seek questions like "What's to prevent a user from copying all the customer records off to removable media?" When these questions come up, explore them, and be sure to keep the discussions civil and focused.

At this point, the best way to proceed is to describe the project to the assembled group. Discuss how the system will function and what services it will provide. You should be sure to list any existing security requirements that are already understood. At this point, run through a brainstorming session to collect any and all concerns that come up. The most important thing is to discuss the issues and enable—encourage even—the team to be as harsh as possible.

Take each security concern the group raises to its logical conclusion, and be sure to understand each one in detail. Make a list, for example, of any preconditions that would need to exist for an attack to be successful. So if an attack would need direct access to a server console, make sure that's clearly annotated in the list of issues the group comes up with.