

1

Secure by Design

WHAT'S IN THIS CHAPTER?

- Understanding your application's security needs
- Discovering the threats to your users
- Identifying potential vulnerabilities

As with any other class of bug, addressing a security issue becomes more expensive the longer you wait to fix it. If there's a problem in the design of the application, trying to fix it in a bugfix release will be very costly because you'll need to change multiple classes. You'll also need to understand and account for the myriad uses and configurations your customers have in place, and be ready to support migration of all these to the new, fixed version of the application. Addressing the issue the first time around means not only spending less time on the issue, but also avoiding the additional (direct and indirect) costs of a security vulnerability out in the field and coordinating a fix release. I once worked on a project for which we spent about three weeks addressing an issue that had been caused by a bad choice of file system path in the planning phase, some years earlier.

Of course it's not going to be possible or even desirable to identify and fix every single vulnerability before writing any code. That's a recipe for spending a great deal of money and taking a very long time to get to market, by which time your competitors will have gotten their apps to the customers. There is a principle software engineers have borrowed from economics called the Pareto Principle, also known as the "80/20 rule." The principle says that 80 percent of the observable effects in any situation are often the result of only 20 percent of the causes. It's a good idea to follow the 80/20 rule in software design — addressing only the most important issues so that the product is of a high enough quality to ship. Which of course leads us to the question, "Which are the important issues?"

ABOUT COCOA SECURITY

Users of your application do not think about what technology was used to create it — whether it was written in Cocoa, Carbon, or Java. What they care about is using the app on their iPhones or their Macs to get their work done. Similarly, their concerns regarding security come not from Cocoa-specific features or issues, but from how the application’s security helps or hinders them in doing their work (or, in the case of a game, in having their fun). Your model of the important security considerations in your app will therefore be largely technology-agnostic, although there are vulnerabilities specific to Objective-C and Cocoa, as discussed in Chapter 9, “Writing Secure Application Code.”

The particular capabilities and APIs available in Cocoa and Cocoa Touch applications become more relevant when you determine how some of the threats you identify might be exploited by an attacker. Cocoa applications on the Mac are part of a multi-user system, as explained in Chapter 2, “Managing Multiple Users,” so understanding how the different users can interact through inter-process communication or by sharing files on the file system will help you decide whether particular interactions could lead to one user’s threatening the security of another. Prioritizing security issues will always be based on an understanding of what your users are trying to do and how that fits in with their processes and with the environment.

You must also have Cocoa-specific features and technology in mind when deciding how to mitigate the threats that attackers may be posing to your application and its users. Users will expect your app to behave in the same way as others on the Mac or iPhone platform, which can be easily achieved if you adopt the frameworks Apple provides for the purpose. As an example, if your application stores a user’s password, he will expect it to use the keychain to do so, because then he can change the settings for that password in the same way as for all his other applications. Keychain Services are described in Chapter 5, “Storing Confidential Information with the Keychain.”

We must now leave Cocoa behind temporarily for the rest of this chapter, as we discuss the principles of application security and discover the threats your users will face as they use your application. These are threats that will be present however you choose to write the app.

PROFILING YOUR APPLICATION’S SECURITY RISKS

To understand an application’s security profile is to understand what risks exist in using that application. That means understanding what your users want to do, what obstacles they might face in getting it done, and the likelihood and severity of those obstacles. Obstacles could come in the form of people attacking the system in some way to extract something of value; honest people could also make mistakes interacting with the application. Either way, a risk is posed only if the application presents the opportunity for the obstacle to upset the user’s work. Figure 1-1 shows how these components go together to form a vulnerability: a possibility that users can’t get their work done safely in the application.

By extension, secure application development is meant to mitigate either the likelihood or impact of those obstacles; you want to reduce to an acceptable level the risk involved in using your app. You can mitigate risk by preventing an attack from occurring, limiting an attack’s impact, or detecting and responding to an attack that is already in progress or complete.



Risk can also be passed on to other entities, usually through insurance policies or service-level agreements (SLAs), but the assumption for now is that this is not an acceptable option, and that you are taking responsibility for the security of your application.

So what presents the greatest risk to your customers? Answering that question is the focus of the rest of this chapter.

Remember that while the discussion is about planning, design, and implementation as if they were separate phases, it's very useful to treat security as an iterative process. Rather than writing a nice security document while designing your app and leaving it to gather metaphorical dust on the hard drive, reevaluate the security model whenever you do a bugfix release or add a new feature. Verify that the issues you prioritized the last time around are still relevant as customers find new uses for your application, or as new classes of vulnerability are discovered and reported by the security community. Have another look whenever a competitor releases a security fix — did you already address that issue, or is your application vulnerable? Automated tests, discussed further in Chapter 9, can be applied just as readily to security practices as to functional testing. You can encapsulate your security assumptions in unit tests or regression tests and discover whether the assumptions continue to hold as the application evolves. Your understanding of the security issues your app faces is therefore critical if you are to have any confidence in the security of your application. This chapter will discuss the creation of a *threat model*, a description of the hostile aspects of the environment your app is in, and of how well the app can protect itself and its users. Maintaining this threat model in electronic form is a must to make it easy to update, and it's a good idea to keep it in the same version control system as your source code so that the state of any version or branch of the application is easy to find. Consider using a database application such as Bento so that the different attackers, threats, and vulnerabilities can be indexed and cross-referenced.

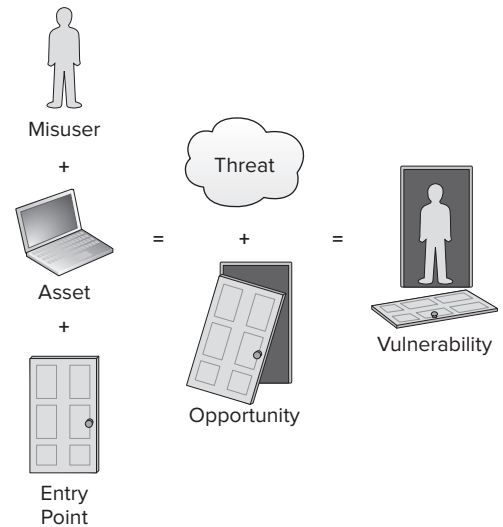


FIGURE 1-1: Anatomy of a vulnerability

DEFINING THE SECURITY ENVIRONMENT

In order to start protecting your application against vulnerabilities, you need to know what the threats to the application are and which are the most important. Before you can even list the threats your application will face, you need to know something about the world the application finds itself in. We can define the application's world in terms of users (both legitimate and otherwise), objects of value that the application holds or accesses, and ways data gets in, around, and out of the application. These are the most important aspects of the threat model, because without a good understanding of the way

the app will be used (and misused) there is no way to be sure that the vulnerabilities you invest time in protecting are important or even present in the app. There is no benefit in protecting against SQL injection attacks, for example, if the application does not communicate with a database.

THREAT MODEL: THYTUNES MUSIC APPLICATION

Throughout this section we'll look at the example of ThyTunes, a free Mac application that allows users to import their own music as well as to buy music from an online store. Music can be organized in the application's library into playlists and synchronized with the ThyTunes application on an iPhone. Music can also be shared over the network with other copies of ThyTunes. And of course the music can be played. None of the analyses for the ThyTunes application shown here will be complete, so see if you can think of any examples of attackers, assets, or threats that are not covered in this chapter. For each new example you think of, consider how important it is in relation to the listed examples; i.e., how important an attack using your new example would be to the application. Think, too, about whether any of the examples here or any examples you come up with are relevant to your own applications, and about applying the reasoning described here to your own app.

Identifying the Application's Users

You should already have some idea of who will be using your app, since you have a customer for your application in mind and have been designing the user interaction for the benefit of that customer. So who is the user? Think about whether you have different classes of users in mind. If so, which is the most important? What will the users mainly be doing with your application? It is important to understand the users' expectations and needs so that you can think realistically about how they will prioritize different risks, how they will react to security issues, and how well they understand the security implications of working with your app. Knowing how your users are working with your application also leads to a better understanding of the risks they could potentially expose themselves to. Which features of the application will they turn on? Will they leave the preferences window alone, or play with every control available? The answers to these questions will help you concentrate your effort on securing the default and most likely configurations of your app.



It is important that the “out-of-the-box” configuration of your app be secure, because that is the one configuration to which all your users will be exposed. Many may never change it, so enabling lightly used features and options means more work to secure the default state of the app. This leads to a higher chance of vulnerabilities being present in a fresh install of the application — vulnerabilities that affect all your users.

You should also think about how technically competent your user is, in relation both to the use cases of your application and to computers (and networks, if relevant) in general. How much information is it appropriate to give the user about what your application is doing, and how much can be treated as “magic” that happens behind the scenes? Should a decision need to be made about a security-related question, is it appropriate to make the decision on the user’s behalf or ask that user about it? Or should the decision be made automatically based on a configurable default? If the application does it automatically, should the user be notified or can it be hidden? Users will appreciate an application that does not ask them any more questions than it needs to, and will be frustrated by one that asks them questions they do not understand. It is not acceptable to wash your hands of security concerns by saying, “I asked the user what to do, and he chose the insecure option.” Aim to provide a great user experience by taking responsibility for the security decisions — after all, you’re the one who has read this book (and if your competitors haven’t, then you get the jump on them).

Closely related to this collection of questions is the question of how likely you think your target user is to make mistakes while using your application. Will this user carefully consider the meaning of any question you might ask, or just hit whichever button is farthest to the right of the dialog? This depends quite strongly on the environment in which your application is used. If your users sit at desks in offices and rely on the application to do their day jobs, then it’s quite likely that they’ll concentrate on what’s going on in the application and pay attention to everything that happens. On the other hand, if you’re writing an iPhone utility, the user may be in a noisy, distracting environment when using the application. If the users are outside in bright sunlight you can’t even rely on their being able to see the display well, let alone concentrate on it. Their interaction may be limited to launching the app and clicking a couple of buttons before putting the device away again, without worrying about the details.

In that environment users may unwittingly expose themselves to risks by failing to understand or consider at all the consequences of their interaction with the application. In such cases you should be designing the workflow to reduce exposure to risk by minimizing the number of decisions users have to make regarding security, and by making it easier to choose the lower-risk paths through the app than those paths with higher risk.

INVESTING THE USER’S MENTAL EFFORT

An example of where *mental investment* by users in security matters becomes important is in dealing with the Internet-based attack called *phishing*. In phishing, the attacker tricks a user into visiting a web site that looks as if it offers some useful service but that is actually under the attacker’s control, such as a site that looks like the user’s online banking service. The user enters account and identification information into the web site — information which is actually sent to the attacker, who can now use it to access the user’s bank account. Users are told to be vigilant, not to click a link without first checking the URL, and not to enter personal information into any web site they’re not sure of. But how reasonable is it to assume that users will do any of these things when they’re using mobile browsers on their phones?

The users of the application may also turn out to be *misusers* — people who can access the application to perform unintended tasks. There is the possibility that a user could accidentally cause a security incident; is there also the chance of a user's going rogue and deliberately attacking other users? An often-cited example is that of an employee who, dissatisfied with the behavior of the company, uses its internal applications to cause it, or specific employees, damage. Such inside attacks can be more damaging than external hacker attacks, as the user already (legitimately) has greater access to the company's facilities and assets. For each of the users you have listed, you need to think about whether this user could become a misuser, and if appropriate to list him or her among the misusers, as described in the following section.

To represent the information you have gathered about your users in a useful and succinct form, create a description of each type you identify. A common technique in user experience and marketing circles, which can help you think about who these users are, is to create a profile of a typical individual in each class of user. These profiles are known as *personae*; for each persona choose a name and even find a photo to put a face to that name. Assign them brief biographies and quotes about why they are using your application and what security concerns they have. Provide each persona with a one- or two-sentence answer to each of the questions asked earlier in this section (along with any others you think could be important for your app). This will help you to reason out the needs and wants of the users by letting you put yourself in their shoes. If you are using a database to prepare your threat model, then each of these descriptions would be a single row in a table. See the following example, “ThyTunes: User List,” for two sample personae relating to the ThyTunes application described earlier.

THYTUNES: USER LIST

This example discusses two ThyTunes users, Susan and Roger.

APPLICATION USER: SUSAN

A typical user of ThyTunes is Susan, a student at a West Coast liberal arts university in the United States. She is not particularly computer-literate, let alone familiar with the workings of computer music or security — she cannot be expected to look after the security configuration of the app herself, and neither does she have an IT administrator looking after her computer to do it. Susan does not even particularly want to spend much time with the application — she just wants to get her tunes playing and then carry on with something else, such as preparing an essay. Therefore, any time she has to stop and think of an answer to a question about working with the application — particularly a technical question requiring special knowledge — will be a time when she considers whether she really needs to be using ThyTunes at all.

Susan has heard of identity theft but believes this happens “to people who visit the wrong web sites on the net, you know the kind I mean — the kind with pictures.” This makes the ThyTunes developers think that she will be likely to ignore any potential confidentiality concerns when using ThyTunes and its music store. It is likely that she could accidentally allow misuse of her credit card information or purchased music unless the application guides her away from risky actions.

MUSIC INDUSTRY EXECUTIVE: ROGER

A second class of user is important enough to the success of ThyTunes to require serious consideration. This class is personified by Roger, the director of partner relations at Balony Music Group, one of the largest record labels. He doesn't actually use the ThyTunes application but is in charge of deciding whether Balony's music can appear on the ThyTunes music store and of determining what to charge the ThyTunes developers for listing that music. He's very concerned that the whole digital music scene makes it much too easy for people to share music without paying the rights holders. If ThyTunes isn't doing enough to stop piracy he'll take the songs off the catalogue. Being able to sell Balony music is a key competitive advantage for ThyTunes, so the developers must treat vulnerabilities that exploit the store or sharing features of the app to pirate music as business-critical bugs.

Identifying the Application's Misusers

Understanding who your users are has helped you see how they think about security, what their concerns are, and how willing and prepared they are to contribute to the security of your application. You can consider the misusers of the application in the same way. Misusers are those people who would — intentionally or otherwise — compromise the security of your application and your users; they are *attacking* the application. When you discover and list the attackers, it's best to include everyone you can think of. Leave discounting certain attackers until you've gotten enough candidates to usefully rank them in order of importance. Considering an unimportant attacker will waste some time, but failing to account for a likely case will undoubtedly cost time, money, and customer goodwill. Remember, too, that the attackers don't have to play by your rules, so don't discount a potential misuser because you find a certain type of exploitation distasteful. An important class of attack might fall through the cracks if you are unwilling to face the idea that the attacker exists. If your app will be used by a charity or religious group that could have enemies for ideological reasons, you need to consider the threat those enemies will pose.

You should collect two important categories of information in order to build up a profile of the attackers: who they are and what their motivation is in attacking your application and its users.

As with the users, create a persona for each class of attacker you have identified, with a mini-biography addressing each of the points described above. Two such personae are shown in the following “ThyTunes: Misusers” example.

Put the attacker's goal and motivations for achieving that goal in his or her own words — computer security can seem at times like a very abstract universe, so having a name and a face for your attacker will make the threats seem more real. Keep the attackers and the users together in the threat model — in this context, they are both classes of people who could misuse your app, for whatever reason.

Who are the attackers? Referring to the information you have already gathered about your users makes it easier to reason about who might be targeting them. If your application tracks online banking transactions and you expect a number of elderly customers, then there's a reasonable chance that an attack might come from a family member who wants to see how much money is in an account (or even

change the amount). If your application deals with medical information there are a host of people who could potentially want unauthorized access.

THYTUNES: MISUSERS

The following discussion illustrates two distinct types of misusers: Tim and Bob.

THE HACKER: TIM

Tim is an unemployed computer science graduate and longtime ThyTunes user. As he currently doesn't have a job, he'd love to find a way to grab the music from the ThyTunes store without paying. He's got a good knowledge of programming and networking, but has use only of his own computer hooked up to DSL. He definitely wouldn't want to get caught in the act as that could destroy his chances of ever getting another job, but if he is successful at obtaining free music then he's likely to mention it on his social networking page. From there his story would undoubtedly be blogged and could even get into the press, which would not be good for the ThyTunes developers.

THE ACTIVIST: BOB

Bob is a member of Inph0free, an online activist group that believes the traditional record labels like Balony are treating their artists unfairly and stifling creativity. The group wants to show that applications like ThyTunes are fundamentally broken in order to pave the way for the next generation of digital music, which will be friendly toward independent artists. Bob doesn't really know what that generation will look like, but he won't stop until he's found a way to discredit ThyTunes for conspiring with Balony and other record labels.

As mentioned earlier, the users themselves could also be misusers, and could attack the application. Recall the example of Susan, the non-expert user of ThyTunes; do you think that if faced with a question about the app's configuration she would always choose the most secure option? If she does not, and compromises the application as a result, she has become an accidental misuser.

How technically competent are the attackers? Are they just going to try scripted attack "recipes" that they can find on the Internet, or will they have some advanced knowledge of the platform they can use to explore the target system? This is particularly relevant on the iPhone, where a less competent attacker may not know about or be able to use jail-breaking tools to drop their own programs onto the device. It's also related to the question of whether the attacker has the skills or inclination to learn specifically about the workings of your application in order to try to discover a flaw. Less able hackers will look for common problems like bad handling of malicious input but might not find issues that require detailed knowledge of your app.

Another factor that determines the type and severity of attack particular misusers are likely to mount is their adversity to risk. A politically motivated hacker could be willing to stop at nothing in an attack on your user; conversely a *script kiddy* may back down at the first sign of countermeasures. In fact, many remote login services display a banner message such as, "It is illegal to use this

service without permission”; some systems administrators have found that even this simple deterrent reduces the rate of attack. (For more information on how login banners decrease the rate of attack, see *Practical Unix and Internet Security* 3rd Edition [Chapter 10], Garfinkel, Spafford, and Schwartz, O’Reilly Media, 2003.)



The reason I frequently call the attackers “misusers” is so I don’t forget that not all security threats are intentional. The word “attackers” automatically conjures up images of criminals or invading armies; in other words, attackers are the “bad guys.” Considering only the security risks born of malice will mean you miss whole classes of important vulnerabilities, specifically those that in being addressed lead directly to a better user experience.



Script kiddy is a derogatory name for an inexperienced or would-be cyber-criminal who is trying to prove himself as a “hacker.” The name comes from the stereotypical — but accurate — depiction of a teenager with few computer skills, whose ability is limited to downloading and running scripted attacks from sites such as SecurityFocus.

For the type of attacker you have identified, how far do you think this person will go in order to complete an attack? Of course this depends in part on the value of the asset being attacked, which we will investigate in the next section.

The attacks that misusers may perform against your users depend on their goals. The possibility of politically motivated attackers seeking to discredit your users or your company has been described already. The profile of the victims and of the attackers, the publicity value of a successful attack, and whether the attackers intend to inconvenience or destroy their victims will all affect the types of attack that they will consider.

The type and scope of such an attack will differ from a personal “grudge” attack against an individual user. A misuser targeting one user will not be probing for vulnerable systems across the Internet, as he or she knows already who the victim is. On the other hand, such a misuser will be more likely to know personal information about the victim, which could be helpful to him or her, for example in guessing the user’s password.

Don’t forget to include indiscriminate attacks as well as those associated with your application specifically. The traditional view of the script kiddy falls into the first of these categories — script kiddies will be looking to “own” as many different computers as they can to prove to themselves and their online friends how “elite” they are, and will not have any reason for choosing your user’s Mac over any other computer on the Internet. Attackers whose target is you as the application’s developer, or even Apple as the provider of the platform, might use your application to achieve their goal.

What resources do the attackers have at their disposal? Will they try to brute-force a password using a beige G3 PowerMac, or a cluster of multi-core systems? Bear in mind that attackers will find it easy (and cheap) to access large amounts of computing power, whether in the form of a legitimate “cloud

computing” service, a university computing laboratory, or an illegal “botnet” comprising previously compromised PCs. Even if these attackers are not very skillful, could they afford to contract an expert, and is attacking your app important enough that they might consider it? How much time can they dedicate to performing the attack? An employee using a 15-minute coffee break to try to access the payroll system will need to try different attacks from an outsider who can spend all night on the attempt — the initial position of the two is also different, which will affect their approaches and chances of success.

TITAN RAIN: A POLITICAL CYBERCRIME

A particularly famous example of a successful attack with apparent political motivation was code-named Titan Rain by the U.S. government. In 2003, attackers, located in China but of unknown affiliation, hid code on PCs located on U.S. and U.K. government networks, as well as those run by defense contractors, including Lockheed Martin. The attackers stole large numbers of files from the computers, which were transmitted to other systems located in a number of companies around Asia before finally being sent on to China. When knowledge of the attack became public, it became the subject of wide press coverage including articles in *Time* magazine: See <http://www.time.com/time/nation/article/0,8599,1098371,00.html> for an example.

The Assets That Can Be Taken

Notice that in the previous section, each of the example users and misusers had some object, real or abstract, that they were interested in. Susan, the application user, mentioned her identity, even if she didn’t see the relevance of identity theft in the ThyTunes context. Roger, the music industry executive, was concerned about money from sales of the music. Tim, the unemployed hacker, had other reasons for being interested in the costs of songs. And Bob, the activist, wanted to damage the reputation of anyone associated with the traditional music industry. Each of these objects, or assets, is something that the application either controls or can provide access to, and that has some value to somebody. It could be that the asset has some value to the user, to you as the developer, or just to the attacker. It is this value that makes the asset important to protect and gives the attacker motivation to take or compromise the asset.

What Makes an Asset Important?

Leaving software aside for the moment, some things have value to you for different reasons. A heirloom may have monetary as well as sentimental value, while a bank statement contains useful information that makes it a valuable asset. An antique dealer would dismiss your bank statement as a worthless scrap of paper, while a loan arranger couldn’t find out much about your finances from the antique. As objects in the real world can be valuable for different reasons, so the assets in an application have different properties that make them important.

Software security experts like to refer to the important properties of an asset with the acronym CIA, standing for *confidentiality, integrity, and availability*.

- **Confidentiality:** This means that the asset has some value in being kept a secret within the application. Attackers may wish to gain the secret for themselves, as is usually the case with a password — the benefit is that the attacker gets to use the password to access the same services as the victim (perhaps even while posing as the victim). On the other hand, there may be some confidential assets — company financial records perhaps — where a successful attack doesn't just disclose them to the user but makes them public. There are also cases in which both types of confidentiality loss can occur. This was the case in April 2009, when a Twitter administrator's password was discovered. The password was used to modify other Twitter accounts, and was also published to draw attention to the problems inherent in having a weak password. The password was “happiness” in this case.



The problems facing Twitter as a result of the disclosure of its administrator's password are described by the security firm Sophos at <http://www.sophos.com/blogs/gc/g/2009/05/01/twitter-security-breach-exposes-accounts-hackers/>.

- **Integrity:** This refers essentially to an asset's accuracy or correctness; what would be the damage were the asset to be modified in some way? Usually there are legitimate changes that can be made: for example, a log file can reasonably have events added to it. It is not desirable for an existing log message to be changed or removed, though. A blog-editing application should let users change their own previous blog posts, but not those of another blogger, unless he or she permits it.
- **Availability:** This property simply means that an asset can be used for its intended purpose. An attacker could stop the victim from using the application as intended; what would be the impact of that? Hijacking is, in a sense, related to availability; the assets are supposed to be available exclusively to the legitimate users. Threats that target the availability of an asset are frequently referred to as denial-of-service (DoS) attacks, as we shall see in the next section.

For each of the assets you identify in your application, think about whether any of these three properties is important. If so, how important are the properties with respect to other properties, and to the other assets on the system? If you could act to protect either the confidentiality of a password or the integrity of the billing process, which would you work on first? The importance of an asset may not entirely be your own decision; there could be laws, industry codes, or other regulations you must take into account. A billing process is a good example, since security standards such as PCI-DSS may be mandated, depending on your customers' requirements or those of your marketers, who may see compliance as a “feature” offering competitive advantage. (The PCI-DSS — Payment Card Industry Data Security Standard — is described at https://www.pcisecuritystandards.org/security_standards/pci_dss.shtml.)

Also think about which of the users you have identified should be able to use the asset during normal operation of the app — is it something users can access, or is there an administrator to take care of it? Does it ever get shared with a remote service, or a user on another computer? Without knowing what the legitimate uses of the asset are, it's impossible to know what uses are illegitimate.



The CIA properties should not be considered exhaustive. If there is an attribute of an asset in your application that doesn't seem to fit in any of these categories, it is important that you make note of that attribute rather than ignore it because it doesn't "work." Failing to keep track of an important asset in your threat model will result in unaccounted-for vulnerabilities in your app.

Exclusivity of an asset is another aspect that could be important in your threat model. A user's Internet connection is not confidential in itself (though the traffic certainly may be), but the user probably has an expectation that it's there to be used as desired, especially if the user is paying for it. An attack in which the connection is hijacked to send spam e-mails shows that this expectation has not been met.

Types of Assets

Many assets are tangible objects, often with meaning outside the application's context. Credit card records are good examples of this type of asset. They have some obvious presence in the application, often having been entered by the user through a form. They have an obvious relevance and value to the user, since the credit card information represents a way for the user to give other people access to the user's money. In fact, because the credit card information can be readily identified with a physical object — the real credit card — it's very easy for users to think about what value they place on the data, because they know what value they place on the real card. Payment records, similarly, are similar to sales receipts in the real world, so it's clear why the vendor and customer would want an accurate record of the transactions that have taken place. Customers want to track what they have spent and ensure that they have been charged the right amount for the right products; the vendor needs to have an accurate record of sales to see turnover and to pay the correct taxes.

A component of the application itself could be a valuable asset. If your application has a logging facility, which audits important actions (see Chapter 7 for more on auditing important operations), then the availability and integrity of that facility itself are important after an attack, for forensic purposes, in tracing the steps taken by an attacker. The attacker would therefore have a reason for targeting the logging component, even though it has no value outside the application.

Sometimes an asset is not valuable in itself, but can become targeted because it provides access to a different asset (or assets). Passwords are included in this category. The password itself is not useful information; it takes on meaning only when it is used to identify the user to some application or service. So an application that does not afford appropriate protection to the password is really giving up access to the user's identity. The value of the password is dependent on the value of the identity — a password for a one-click purchasing system has value equal to that of the money that could be lost through the purchasing process. It also follows that when users access multiple accounts with the same password, that password takes on a higher value — which is the reason for using a different password for each service. Of course it is the confidentiality of the password that is most important; the identity is protected by the password only when the password is secret.



Notice that assets like a password or private key can exist only when you have determined that something else needs protecting. There is no point in password protection that keeps people out of nothing. Similarly, requiring a password to access non-confidential information would be considered just annoying. So to determine that the application needs to have a password-protected component, you need to have already decided that something needs to be protected and that using a password is a good way to provide that protection. This demonstrates that a threat model must be considered iteratively — each countermeasure designed into the system certainly changes the security profile of the existing system. But it may not merely mitigate some existing risks; adding a countermeasure such as a password may introduce new threats that must be taken into account.

Still other assets do not have any tangible presence in the application, but can nevertheless be of value. Bob, the information freedom activist, wants to damage the reputation of the companies behind ThyTunes and its services. The reputation isn't a part of the application, and it isn't even stored on the computer. Nonetheless it has some worth and the application can be used to compromise it. But why would any attacker want to target your reputation? Possible reasons include:

- **Political motivation:** Perhaps your company is based in a country that has poor relations with the attacker's country.
- **Competitive discrediting of your company:** Your reputation can also be indirectly compromised by an attack targeting some other asset, simply because the attack was possible.

Any successful attack is also likely to damage that other important intangible asset, the confidence of the victim and your other users and customers.

Intangible assets can be the hardest to think of because they are often nebulous concepts, bearing only an indirect relation to the software and its purpose. Reputation and user confidence are great places to start thinking, though, because if these lose their value then it becomes hard to sell any software. At the time of writing, the online store for a popular browser maker has been unavailable for over a week since a security breach was discovered in the software running the store. In addition to the direct cost of lack of sales, how many people will question using the store once it does reopen? How many will question the security of the browser software?

There are also assets that are incidental to the application entirely. A commonly misused asset in many modern attacks is the network connection of the host computer. The network connection isn't "owned" by any application of course, but an attacker can use a vulnerable application to turn the computer into a "zombie" that connects to other systems, perhaps as part of a spam or distributed denial-of-service (DDoS) campaign. The attacker could also want to access the CPU time of the zombie computer, perhaps to try to crack an encrypted password. Remember that just because the asset isn't "yours," that doesn't mean the attackers won't use your app to get at it — they don't need to play by the rules.

THYTUNES: IMPORTANT ASSETS

The user's music in the ThyTunes application is the asset with the most direct relationship to the use cases. If the music were to be rendered unplayable, then the user would not have any reason to use the application. If it were to be made publicly available, the record companies would lose sales. So the user is concerned about the availability of the music purchased, but the record label director is concerned about the confidentiality of that music.

Because users are able to purchase music through the ThyTunes store, there must be some way for them to present payment information, such as credit card authorization, to the application. Users will want to ensure that the confidentiality of their information is protected, so that their own credit cards cannot be used by other people. The developers of ThyTunes want to be certain of the integrity of the payment information, to avoid problematic disputes over questioned payments. The developers are also concerned over the availability of the payment-handling service, since any time this service is unavailable they are unable to sell any music.

Were there to be any public security problem with ThyTunes, the record labels would question their partnership with the developers. They see the application as being primarily about the relationship between the users and their music — and thus the labels as providers of that music are going to be affected by any damage to the user's trust. Any damage to the reputation of the ThyTunes developers as providers of a secure and trustworthy application could have a direct financial impact on the record labels' business.

Understanding the Data Flow

So far, to borrow the jargon of TV crime dramas, you have profiled your attacker by identifying and understanding who the misuser is. And by identifying the valuable assets in the application you've given the attacker a motive. You also know who your users are, i.e., who the victims would be in any attack. In order for a threat — or potential attack — to exist, there needs to be a *modus operandi*. There must be some way for the attacker to get to the asset to steal or compromise it.

Imagine that you are in charge of security at a museum, and have been tasked with securing a priceless diamond from theft. You would need to know where all the doors, windows, air-conditioning vents, and other ways to get into the building were. You would also want to understand how the would-be thief could get from any one of these entry points to the diamond, and then back out of the building. So it is with computer security: you have found your jewel thief and the next gem on her target list, now you need to know how she'll break in and get back out with the diamond.

Applications have entrances and exits just as buildings do. In the software case, these are any places data goes into or out of the application.

The User Interface

The user interface is a good place to start. Of course, the person using the interface is the user, isn't he? Well, not necessarily. All you really know is that the person using the interface is the person who currently has access to the keyboard and mouse. But if the user has left the computer and lets somebody else take control of the application, isn't that the user's problem? If you could simply deal that way with any of the security issues you encounter, now would be a good time to put the book down and release the app. Instead, you might consider the following possibilities:

- The user accidentally did something unexpected.
- The user was coerced into doing something unexpected.
- Someone deliberately misused the user interface.

Apple's Mac Human Interface Guidelines already give you a way to deal with the user who performs a potentially insecure operation, by creating a forgiving user interface. The guidelines (at <http://developer.apple.com/documentation/UserExperience/Conceptual/AppleHIGuidelines/XHIGHIDesign/XHIGHIDesign.html>) say:

Encourage people to explore your application by building in forgiveness — that is, making most actions easily reversible. People need to feel that they can try things without damaging the system or jeopardizing their data. Create safety nets, such as the Undo and Revert to Saved commands, so that people will feel comfortable learning and using your product.

Warn users when they initiate a task that will cause irreversible loss of data. If alerts appear frequently, however, it may mean that the product has some design flaws. When options are presented clearly and feedback is timely, using an application should be relatively error-free.

Anticipate common problems and alert users to potential side effects. Provide extensive feedback and communication at every stage so users feel that they have enough information to make the right choices.

These are good ways to deal with either of the first two UI issues in the earlier bullet list, because when presented with the question of whether they want to go ahead with a damaging task, the users should (you hope!) ask that very question of themselves. However, as noted by Apple, if you ask repeatedly for confirmation it will be frustrating for the users. If you look at the interface for your application and decide that there are a number of different actions that should all be confirmed before they are carried out because they increase the risk to one or more assets, then perhaps the application should not permit those actions at all — i.e., they should possibly be removed from the user interface. That would certainly make it easier to protect the assets. There are other ways to communicate concerns to the user, including a warning message alongside the



FIGURE 1-2: Configuration pane describes system's security status.

UI control that introduces the risk. The firewall configuration pane of System Preferences, shown in Figure 1-2, provides a good example: it allows the user to turn the firewall off without confirmation, but describes the security state of the system in a message on the pane.

Even the third possibility in the bullet list — that someone will abuse the application intentionally through the user interface — can be dealt with. On the Mac you can distinguish different users with varying levels of access to the application, and provide for a user who wants to perform some privileged operation using Authorization Services (see Chapter 2 for more on managing multiple users, and Chapter 6 on performing privileged tasks).

The iPhone does not have such a concept of multiple users, so the user is capable of performing any action that is available in the UI. This leaves three options:

- **Do not put that action into the UI:** This may be appropriate if users can get along without needing to perform the action, or if there is some alternative way to get at it. For example, you may not want to allow users to delete an account on a web service through the iPhone application, if they can log in at the browser to do that.
- **Provide your own “escalation” mechanism:** A good example is the iTunes application on the phone, which always prompts for the user’s password before a purchase, even though it could store the password in the keychain. By forcing the user to re-enter the password, iTunes ensures that the escalation from “handset holder” to “iTunes account owner” is performed by only the correct person.
- **Documentation:** Explain to the user the importance of having the whole phone locked and PIN-protected through the Settings application when your app is installed, even when the user isn’t using that app. This option is the least convenient for the user and is unlikely to work (will the user actually read your recommendation?), so its use should be considered carefully.

Data from External Sources

Not all the data in an app is necessarily going to come from the user working with the interface. Data could be coming from an external source such as a web service. Consider the large number of agencies involved in getting data from the web service to your application:

- The user’s local network (which may not be the home network, of course; the user could be using a wi-fi access point in a hotel or coffee shop)
- The user’s Internet connection
- Various domain name service (DNS) providers
- The network to which the web service’s server is connected
- The various networks on the Internet between the user’s network and the web service’s network
- The web service provider
- The other users of the web service

It would be a very trusting soul who believed that none of these agents could be hostile. It pays to be very careful when dealing with remote data, as the outside world is the natural home of the attacker.

While there is, as we have already discussed, a chance that one of your users is misusing the application, there are plenty more potential misusers among the billions of other people on the Internet. How does the information that comes from remote sources get handled in the application? What components does it get passed between? Which assets do those components give access to?

If data of a certain type coming in through a particular entry point eventually causes problems in a certain method from a class in an entirely different module, then attackers could use that problem to their advantage. But such issues can be hard to track down, as it may not be clear that the other module is ever used during the handling of that data. The interfaces between components should thus be listed, as well as interfaces that are purely on the application's "surface," such as its connections to network resources. If a Mac application launches a background task or communicates with a daemon, then what can happen if the task is launched outside the app or the daemon is sent arbitrary requests? Is a string that it is safe to use in the network-communications component still acceptable when it is passed to the database module? Because the protocols used are different, the techniques that should be used to guard against misuse of each component differ. This reinforces the idea that every component should either mistrust what it is told, or should "trust, but verify."



The phrase "Trust, but verify" has been attributed to various people. I was first told the phrase by a manager who attributed it to Soviet leader Nikita Khrushchev. Since then I've seen it attributed to at least two U.S. presidents, as well as the writer Damon Runyon. And it's also been reported as a translation of a Russian proverb. Whoever said it first, it's good advice — as well as an illustration of how hard it is at times to verify something.

Such boundaries can include interface calls between different classes, particularly to or from loadable frameworks or bundles. The questions to ask here are which of the assets in the application can be accessed directly by the bundles, and which can be accessed through the application programming interface (API) exposed to the bundle — that is, whether through the published interface or as a result of the bundle's calling undocumented methods. The goal is to know everything that happens once some data enters the application — or at least to understand enough about where everything happens with that data to make reasonable guesses about which assets are at risk from abuse of that entry point.

Also think about how the application behaves if the interface is misused — for example, if the bundle calls methods out of sequence, or uses unexpected values for some parameters. How could your application deal with these cases? If it is possible to clean up the malformed parameters and carry on normally with the sanitized versions, this could allow the application to recover from some genuine mistakes including accidental bad input from the user or a bug in the bundle code. On the other hand, if the bundle really is going out of its way to misuse the API by calling methods out of sequence or with clearly malicious data in the parameters, then it could be best to avoid trying to honor its requests. Techniques for doing this range from ignoring the request (perhaps noting in the log that this was done) to aborting the whole application. That last suggestion seems quite extreme, but if it is detected that misuse of the application has already caused it to get into an inconsistent state with no hope of recovery, it could be the only safe measure to take. Attempting a more sophisticated form of recovery when dealing with inconsistent or malicious data could end up getting the application into a worse state than simply giving

up and stopping, distasteful though it may seem. Some very nasty vulnerabilities have resulted from developers — particularly of web browsers — trying to be too clever in accepting malformed data.



There are cases in which the operating system will make the decision on your behalf that the security of your app has been irrevocably compromised. It will then kill the app without any alternative action offered. One example is the case of the stack canary, a guard against attackers modifying the way an application exits from functions. Stack canaries in OS X are described by Amit Singh at <http://www.kernelthread.com/publications/security/smemory.html>, and in Apple's Leopard Security briefing: http://images.apple.com/au/macosex/leopard/pdf/MacOSX_Leopard_Security_TB.pdf.

Coding for Security

It sounds as if the treatment of data entering, moving through, and leaving the application depends very specifically on the app's implementation, so shouldn't it be dealt with at the code level while you write the app? Yes, but there are good reasons to understand the data flow while designing the app, too. You're more likely to be thinking about the "big picture" and how the whole app fits together when designing it than when coding it, when you will be dealing with the specifics of individual classes or methods. This all means that it's much easier to see where the data is going while working on the design, and to identify relationships between distant parts of the app that might go unnoticed while you're knee-deep in the code. For example, your big picture view might identify sensitive data such as a password or some confidential data. Identifying the flow up front may enable you to define a strategy for consistent protection of this data — it's common for applications to encrypt data but forget the other points of access including the network service for which the password is used. Even a seemingly small application with only a few use cases can comprise several components.

SOME OF THE THYTUNES ENTRY POINTS

The external interfaces for ThyTunes include the Mac and iPhone user interfaces, which allow users to buy music through the store and organize their music libraries. Organizing these libraries includes the ability to delete music. The store is accessed over the Internet via a web service. Additionally, the Mac and iPhone applications communicate over a local network connection to synchronize music and playlists.

I have not listed any internal interfaces in the applications, because they really depend on the way the components in the application are hooked together. This is where knowledge of the specific design of the application will be necessary to work out where these interfaces are and how important each is.

It can be very useful to annotate any sketches or other design documentation you create with comments about the source and safety of data or callers, to act as an aide-mémoire when you (or another developer) write the code. If your design includes a skeleton of the source code, then writing these notes in code comments alongside the class or method descriptions is best; it will be much harder to ignore or forget about the security notes if they are staring back at you when you open Xcode. Such notes could explain who might have prepared the data that's being handled by this code, whether any change of privilege is involved, and any particular sanitization that you should deal with before using the data. For instance, a method for writing a Twitter message into an SQLite database would have an associated comment explaining that the message could have come from anyone on the Internet but will be in the privileged position of modifying the database asset, and should be cleaned of SQL injection attempts before being used.

DEFINING THREATS

Returning to the crime show analogy, you now know the victims, perpetrators, motives, and modus operandi. If any one of the perpetrators could use one of the MOs to achieve a goal, then there is the potential for a crime to be committed. The threat of crime hangs over your application.

What Is a Threat?

A threat is the risk that one of your attackers could use an entry point of the app in order to damage or take the asset the attacker willfully or accidentally targets. As with any risk, there are two important dimensions: the likelihood that this threat will actually occur and the impact, or amount of damage, that would happen if it did occur. There's no reason to go into very fine-grained analysis of either impact or likelihood; ratings of low, medium, and high will be sufficient to compare the threats with each other and identify those that should be dealt with first.

Visualizing the Threats

A useful way to visualize this information is on a graph of impact versus likelihood, with each threat plotted as a point on the graph. The risk you are willing to accept, or that you believe your users would be willing to accept, is drawn as a line separating the top-right and bottom-left corners of the graph. The more threats toward the top-right corner, the higher the risk the application is exposed to. When you have that graph, which will look like Figure 1-3, congratulations! You've now modeled the environment of your application in terms of the risks it will face. This is the first important milestone in the complete threat model — or, more correctly, in the initial version of the threat model. As the application changes, and features are added, bugs

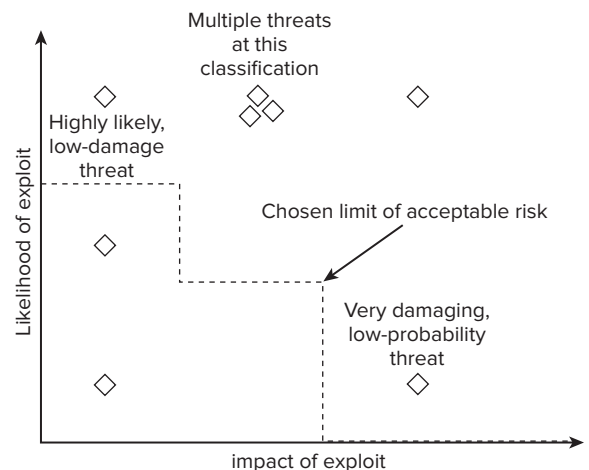


FIGURE 1-3: An example of a threat graph

are found, and countermeasures are designed in, the threat model should be updated to reflect the current state of the app. A static threat model representing the 1.0 release (or worse, the pre-alpha version) of the software you released two years ago will not help you understand the security of your application as it evolves.

It isn't strictly accurate to depict likelihood and impact as independent properties; if attackers can cause a lot of damage to a system in one way, they may be more inclined to attack in that way. That really depends on the profile of the attacker: if a vulnerability is going to be targeted only by script kiddies, then the attackers probably lack the skills or aversion to risk required to mount a continuous campaign to realize one particular attack. Similarly, if a threat is very likely to be realized, then the level of damage may come not just from the direct attacks on the target asset, but from the negative reviews of an app that is continually suffering from security problems.

For the first evaluation of an application's threat model, it should not be assumed that any countermeasures have been designed or implemented in the app. If the countermeasures already exist, then the tail is wagging the dog; the threat model is supposed to show you where to expend effort in protecting the app. If you assume that some countermeasures are in place before constructing the threat model, then you aren't considering the importance of the threats if those countermeasures are not in place. Thus you can't be sure that the countermeasures are truly the most effective, nor that it is a good use of your time to implement them. List and rank the threats first, then identify the vulnerabilities (see the next section) that should be protected against. Design countermeasures into the application to deal with those vulnerabilities. Now look at the threat model again, see how the countermeasures have mitigated some threats and created others, and draw the graph for the new state.

Similarly, do not worry yet about the specifics of how the attacker will perform the attack, as this could depend on details of the application that either do not yet exist or are inappropriately specific for a design-level analysis. The reason you are identifying threats is to consider what attackers might be trying to do to the application, how bad it would be if they managed to do this, and what the likelihood is of their pulling it off. The exact set of circumstances through which the attack might be successful constitutes a vulnerability, discussed later in this chapter.

Assessing a Threat's Likelihood

It is not going to be easy to assess how likely any of the threats you identify may be. If you are building the threat model while designing the application, then the app has not yet been exposed to the real world and so none of the threats has yet come to pass. Of course, if you are working on a threat model of an existing application, then you may already have had to deal with some security issues. Attackers (particularly unskilled ones) are likely to try variations on a theme, thinking that you have addressed only the specific problem revealed last time. Or maybe they just lack the skill to think of something different to try. So they might attempt the same attack on a different entry point, to see if the same problem exists elsewhere.

Unfortunately, victims of information security attacks — and the software vendors who supply them — are often reluctant to publish information about the attacks they have faced, though you might get lucky and come across details of vulnerabilities found in competing or related apps (a vulnerability is an unmitigated threat, and will be covered in the next section). The probability is that an attacker will look for vulnerabilities discovered in other, similar applications to use as a

template when targeting yours — therefore the likelihood that the threat will affect your users in addition to your competitors’ is increased.



A final way to determine the likelihood of any threat’s coming to pass is to look in general at vulnerabilities reported in any other applications and try to discover trends. The U.S. government’s National Vulnerabilities Database at <http://nvd.nist.gov/> is a good place to look for current and historical security problems in well-known software, as is the SecurityFocus web site, <http://www.securityfocus.com/>. SecurityFocus also hosts the Focus-Apple mailing list (<http://www.securityfocus.com/archive/142/description>), a low-traffic list discussing “security involving hardware and software produced by Apple or that runs on Apple platforms.” Apple announces security advisories for its own products on the security-announce mailing list at <http://lists.apple.com/mailman/listinfo/security-announce>.

Once all the available facts have been exhausted, you must compare the relative likelihood of distinct threats using judgment or even intuition. Some are simple to dismiss; if a particular attacker has no physical access to the computer, then any threat performed by that attacker that requires physical access is unlikely. Consider the values of the assets that are threatened and how likely they are to be associated with each route into the system. If it would be obvious to people that a particular asset is accessed through a given entry point, then threats targeting that combination of asset and entry point seem likely. Also think about how many of the relevant class of misuser you might expect; a particular type of threat might be more common simply because there are many people who might try to perform the attack.

The impact of each threat is easier to evaluate than its likelihood. We have already considered the relative importance of each asset and which of the CIA properties is important. We can similarly classify each of the threats to understand how it affects the system (see the next section, “STRIDE classification”). If a threat is going to cause an important compromise of a valuable asset, then it is more important than a threat affecting only inconsequential assets.

STRIDE Classification

Simply knowing that a threat exists is not sufficient information with which to plan countermeasures to reduce or mitigate the threat. You also need to understand what it is the attacker gains (or you or your users lose) by any particular threat’s being realized. The different possible effects that the threat can produce in the system are known by the acronym STRIDE. The categories in the STRIDE classification are not graded in the same way as the likelihood of a threat or the availability of an asset; they are yes/no properties of the threat. A threat either represents the possibility of realizing one (or more) of these effects or it does not. The effects are:

- **Spoofing:** The possibility that an attacker will be able to pass him- or herself off as a different person. An external agent might be able to perform a task as if the application’s user had performed it, or one user could appear to be acting as a different user.

- **Tampering:** The realization of the threat would result in the modification of an asset that the attacker does not normally have the right to modify, or just not in the way made possible by this threat.
- **Repudiation:** It is not possible to demonstrate that the attacker accused of realizing a threat was really responsible for the attack. This can be an important property of a threat if an application must meet some legal or regulatory security requirement, like applications that process online payments. Such requirements are also common for corporate software systems. Without the ability to prove beyond reasonable doubt who the attacker is, it is impossible to place sanctions on the attacker, through either legal prosecution or corporate disciplinary measures.
- **Information disclosure:** The attacker has gained access to some confidential asset that this person should not normally be permitted to view.
- **Denial of service:** The availability of an asset has been compromised. There are many ways in which this could occur. The asset could be removed (which is often also a tampering event). It could become impossible to reach the asset; perhaps it is located on a remote system and the network connection has been rendered unavailable. The asset could also be slowed down or made so busy as to refuse or infrequently allow legitimate access; this is how the classic DDoS on the Internet works.
- **Elevation of privilege:** The attacker is able to perform some action that is not normally permitted for someone with the attacker's privilege level. This can often be the result of a spoofing attack, though that is not necessarily the case. Once an attacker has elevated his privilege level, he usually gains the ability to carry out attacks from any of the other threat classes listed here. Security analysts therefore treat privilege escalation as a very serious class of threat.

THYTUNES THREAT DISCOVERY AND CLASSIFICATION

Some of the threats that can be identified from the environment of ThyTunes are described here. The list is not complete but shows how knowledge of the app's environment can be used to consider the sort of attacks that might be likely. Figure 1-4 shows the three threats in a graph as described earlier. It should not be much of a surprise that the threats identified here all lie toward the top-right corner of the graph, as either high-impact or high-likelihood threats. By identifying the most important assets, attackers, and entry points you make it easy to selectively aim at the more threatening attacks the application will face.

The ThyTunes developers have decided that any low-impact or low-likelihood risk can be accepted, and the risk from any other threat must be mitigated. The amount of risk they are willing to accept is shown in Figure 1-4 by the dashed line, which shows that they will need to address each of the three threats discussed after the graph. Try to think of more threats that would be present during the use of ThyTunes, and place them on the threat graph relative to the threats shown. Will the developers need to mitigate all the threats you have identified? Do you agree with their choice of acceptable risk level?

DELETING ITEMS: DENIAL-OF-SERVICE THREAT

A regular user of the application, either at the Mac or iPhone interface, accidentally deletes purchased music while organizing the library. This is a denial-of-service threat in the STRIDE classification, as the availability of the purchased music has been compromised. Based on the profile of the users defined above, this threat would seem very likely. It would cause a single user severe problems, basically stopping this user from using the app for its intended purpose. But since it doesn't affect multiple users, the developers choose to rank its impact as medium.

STEALING SERVICES: CONFIDENTIALITY THREAT

The out-of-work attacker or the online activist targets the web interface to the online store in order to retrieve music without paying. This would probably be categorized as affecting the "confidentiality" of the music, since the record company sees the music as being licensed to legitimate customers when they purchase it and does not want it to be available to anybody else. This attack would also indirectly damage the relationship between the ThyTunes developers and the record company, which doesn't want record piracy to be possible in the online music world. This vulnerability could be exploited, allowing attackers access to information they should not have, and could potentially be a spoofing attack if the attacker poses as a paying customer. It's also a privilege escalation threat as the music is supposed to be available only to people who can pay for it. If one user found a way to purchase songs on behalf of all other users, the whole system could be thrown into chaos. The customers would expect refunds of all the spoofed purchases. The developers of ThyTunes could expend lots of effort in identifying all the attacker's orders and reversing those. They could refund every purchase made during the attack or insist that the payments stand. In any case, the story is likely to get into the press and damage the company's reputation with both ThyTunes users and with the record companies.

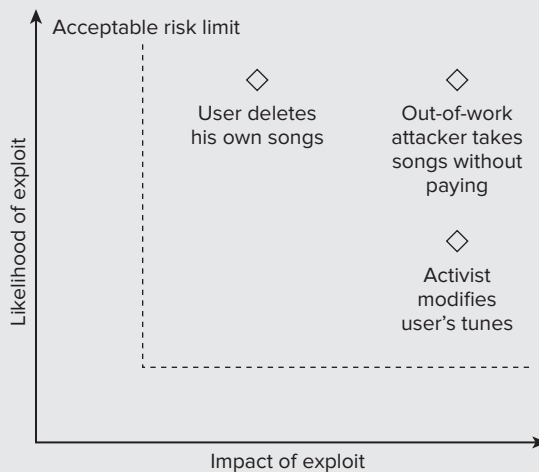


FIGURE 1-4: Threat graph for the ThyTunes application

(continued)

The likelihood of this threat's being realized is probably high, as there are many attackers who might try it, and the end result — free music — is appealing. The potential consequence, that the record company withdraws its music from the ThyTunes store, strikes at the core of the ThyTunes business, so the expected impact is also high.

CHANGING THE INTERFACE: TAMPERING THREAT

The online music activist uses the library syncing network interface to replace tunes in a user's library with messages about the activist group's mission. This would attack the integrity and the availability of a user's library, and as with the previous threat could tarnish the developers' relationship with the record company (which is, after all, the real goal of the activist). This threat is classified under STRIDE as a tampering threat. It is also a privilege escalation threat, as the syncing interface is supposed to allow users to sync only their own computers and phones. As with the previous threat, the expected impact is high, since there is a threat to the business. It is not thought that there are many of these activist attackers with knowledge of the syncing mechanism, but the profile of such attackers suggests that they would expend some effort to attack the system. The likelihood of attack is therefore medium.

So now that you know all the constituent parts of a threat, and how to classify threats and rate their importance relative to each other, you can now start to consider countermeasures to mitigate those threats — reducing their likelihood, impact, or both to an acceptable level. The remaining chapters in this book all discuss aspects of Mac OS X or iPhone OS technology with a view to creating technical countermeasures, and as such will make use of the terminology and ideas described in this chapter. However, before we leave the world of the on-paper application design behind, there is one remaining aspect of an attack to consider. You have the victim, the perpetrator, the motive, and the MO — for the complete criteria for a crime the attacker needs an opportunity. This brings us to the unmitigated threat — the vulnerability.

Classifying the threats can also help to rank their relative importance. Recently there have been a number of news stories about companies, universities, and government departments losing personal information about customers, members, or the public, so information disclosure threats could resonate particularly with individual users. Elevation of privilege is usually seen as a critical threat, especially if it gives the attackers “root” privileges on the victim's computer, which lets them do anything with that system, including sending spam messages and using it as a base for further attacks against other victims.



Examples of the impact information disclosure has in the media can be found at <http://www.darkreading.com/insiderthreat/security/app-security/showArticle.jhtml?articleID=219400878>, http://www.theregister.co.uk/2009/08/21/trade_body_data_policy/, http://www.dailycal.org/article/106339/campus_takes_steps_to_boost_server_security_after_, and <http://www.guardian.co.uk/uk/2007/dec/11/northernireland.jamessturcke>.

DEFINING AND MITIGATING VULNERABILITIES

The previous section was about defining threats faced by your application: potential misusers and the ways in which they could target the app. This modeling of the hypothetical hostilities in the app's environment enabled you to balance risks against each other to find the most important ones, but of course it's only the ones that really happen that will cause problems when the app is deployed. If there exists a threat against which the app is unprotected, then the app is said to be vulnerable to that threat. The circumstances that result in the threat's being exploitable comprise a vulnerability.

Vulnerabilities as Unprotected Threats

For each of the threats you have identified as significant, try to think about how the attacker might go about targeting the application — how the threat could be realized. Some vulnerabilities are so common that tools have been written to automate their exploitation (which has the side effect of automating the detection of those vulnerabilities, too).



A discussion of a popular set of penetration-testing (or pen-testing) tools can be found in Chapter 5 of OS X Exploits and Defense by Kevin Finisterre et al., Syngress Publishing Inc., 2008.

The definition of the vulnerability is essentially a “use case” for the attacker's interacting with your application; perhaps a better term would be a “misuse case.” A convenient way to describe use cases is with the user story, and that's also a handy way to describe misuse cases. We need to go into more detail than is usually present in user stories. (User stories are explored in detail in *User Stories Applied: for Agile Software Development* by Mike Cohn, Addison-Wesley, 2004.)

Common practice when writing user stories is to ignore, or rather actively avoid discussion of, the application's implementation. When writing a description of a vulnerability, you need to describe how the attacker is going to exploit the application in order to carry out the attack — that means that you must include details of what makes the application vulnerable. A complete misuser story must therefore describe which threat is being realized by which misuser, and what this misuser does to the app to make the threat happen. The amount of detail you go into is entirely up to you. The description should be sufficient to help you remember what the vulnerability is about — either when working on its mitigation or when discussing it with others — but not so long that either the security model becomes unmanageable or you spend inordinate amounts of time on creating and maintaining it. The sentence “Attacker uses SQL injection in the recipes field” could be a useful aide-mémoire, or you might want to go into more depth. Sometimes it might be quicker just to fix an obvious bug than to think about all the conditions and write out a description, even as a simple misuser story. You should use your own judgment, and take your company's processes and development lifecycle into account. I have worked on teams with very specific change control mechanisms, designed to ensure that all modifications made to the teams' products are understood and tested. In such situations an ad hoc change would not be welcome even though the alternative is more expensive.

For an app that still exists entirely on the whiteboard, this definition of vulnerabilities will take two forms: discovering and addressing vulnerabilities in the design of the application, and making notes of problems that could come up when you implement the application. Both types should be documented in the threat model. If you make a design choice based on security but fail to explain in the design why you made that choice, you might change the design later without remembering that the original choice was justified as a vulnerability mitigation. If you do not leave reminders to yourself or fellow developers on your team to defend against vulnerabilities when writing the code, then you or they will probably forget to do so. Remember the adage that every application has a team of at least three developers: you from six months ago, you, and you six months into the future. Your design and threat model should accommodate each of these developers.

A user story is complete when the user can complete the task identified in that story (or, more correctly, when the users accept that they can complete that task). The criteria for completing work on a misuser story are more nebulous: the story is completed when you or your customers can live with the remaining risk associated with the vulnerability. The most visible form of that completion is when the risk is reduced to zero; either the vulnerability is impossible to exploit or there is no impact caused by successful exploitation. If it is not possible to get to exactly zero risk, then you will need to decide what constitutes “acceptable” residual risk. The DREAD classification outlined below can help you to understand what makes a vulnerability likely or damaging, and then to decide which aspects of the vulnerability to address in mitigation of the risk.

The number of threats faced by any application may be fairly small; on one project I worked on, three developers brainstorming for a total of two hours ultimately came up with a list of around a dozen threats. Depending on the relative risk associated with each threat and the level of risk you choose to accept, you could easily decide not to deal with half the threats you list. However, when constructing the threat model for an app that is either still in the design phase or has never been the subject of security analysis, it could be easy to come up with a very long list of vulnerabilities. For example, a tampering threat in which the attacker is a local user of the computer implies that each file in the app could be vulnerable to a file-system-permissions or access-control attack, as could named sockets or pipes, temporary files created while the application is running, or listeners on loopback network connections.

The number of vulnerabilities that could appear as a result of the analysis might seem overwhelming. Vulnerabilities can be grouped if they are similar — a vulnerability in which the attacker sends too much data to an entry point and one in which the attacker sends too little data to the same entry point could both be classed as a data sanitization vulnerability of that entry point. The group of vulnerabilities can then be treated as a single issue to be addressed in the application. Be careful not to lose too much information in this process: the combined vulnerability descriptions should still be sufficient to remind you of the problems that must be dealt with. Forgetting to deal with one aspect of a vulnerability still leaves the application vulnerable.

Estimating Risk

As you have already defined a risk level for each of the threats identified, the risk associated with each vulnerability is roughly known. The information you have on the vulnerabilities can be used to refine your understanding of the risk associated with each threat, and the relative risk of each vulnerability needs to be known so that you can prioritize dealing with it. It is, after all, the vulnerabilities themselves that you will be addressing, not the abstract threats underlying them. Remember to take into

account the information you have already gathered about the security profile of the application; will the attacker actually have the competence or tenacity to discover a complex vulnerability? Is a vulnerability that targets the confidentiality of public information worth spending any effort on?

If you were unable to find any vulnerabilities that realize a particular threat, then it's likely that this threat is not exploitable, or at least that the probability of its being exploited is lower than you might have initially thought. Conversely, if there is a huge pile of misuser stories all corresponding to one threat, then the threat is probably an important one to deal with.

Another thing to look out for is outlying vulnerabilities. The DREAD classification described in the next section will help you to identify the likelihood and damage associated with each vulnerability. If there's one that seems obviously exploitable or particularly dangerous, then it could indicate that you've underestimated that threat, and that it's worth trying to uncover other related vulnerabilities (or at least satisfying yourself that there are none). Outliers in the opposite direction, for example a very unlikely vulnerability in a sea of easy targets, can probably be dealt with last if at all.

DREAD classification

The properties by which security analysts classify vulnerabilities are a refinement and augmentation of the two dimensions of impact and likelihood that have so far been used. As with the STRIDE classification for threats, your consideration of these properties can both drive and be driven by the understanding of the importance of the risk associated with the vulnerability. Each of these properties should be given a rank on some simple scale, such as low-medium-high or 1–5. The DREAD classification system was first suggested by Microsoft: see *Writing Secure Code, Second Edition* by Michael Howard and David LeBlanc, Microsoft Press, 2003.

- **Damage:** The severity of the effect were this vulnerability to be exploited in the field. This property is related to the impact of the risk to which this vulnerability relates, as well as to user perception of the exploit. As has already been described, information disclosure attacks are currently garnering a lot of press coverage, so an information disclosure threat should be rated higher than any other threat that would otherwise be considered equally damaging.
- **Reproducibility:** The likelihood that the same series of actions on the part of the misuser will always lead to the vulnerability's being exploited. Evaluation of this criterion includes questions such as whether the vulnerability is time-related: a vulnerability in the default state of the app is inherently more reproducible than one that can be exploited only while certain operations are in progress, or when non-default features are enabled.
- **Ease of attack:** A measure of the simplicity of exploiting the vulnerability. This can incorporate issues like the required technical competence and time or resource investment of the misuser and any environmental prerequisites, such as whether the attacker must have physical access to the computer or phone and the level of privilege the attacker needs to perform the attack.
- **Affected users:** The fraction of the user base that could be affected by exploitation of this vulnerability. Does the vulnerability work only on particular models of the iPhone, or affect only users with specific releases of Mac OS X? Perhaps the application is vulnerable on 32-bit machines but safe in 64-bit mode; how many users have the vulnerable version? If the exploit relies on a custom app configuration, such as a certain feature's being enabled, how many of the users are likely to have that configuration?



Note that ease of attack and reproducibility are not necessarily linked. A vulnerability that requires 72 hours of manual preparation and a cluster of 10 Xserves to exploit but against which an attack always succeeds means an attack that is difficult but 100 percent reproducible. An exploit that can be run from an automatic script but that succeeds only once in every million attempts is easy to perform, but not very reproducible. A vulnerability that can be exploited both repeatably and rapidly can be catastrophic.

- **Discoverability:** How likely do you think it is that the vulnerability will actually be found in the field? This is quite a contentious issue — the authors of *Writing Secure Code* choose to assume that all vulnerabilities will eventually be discovered and assign this property the highest value every time. A cynical person might think that the discoverability is only recorded to add credence to the (probably back-formed) DREAD acronym. I think it is possible to use information about your misusers and the specifics of the vulnerability to answer the question, “Will the attackers find this vulnerability?” and that it’s important to do so to get a better view of the likelihood that a given vulnerability will be exploited. If discovering an exploit would require in-depth knowledge of the design of your application that you don’t expect the attacker to have, then it’s probably going to be harder to find than a “well-known” vulnerability like a buffer overflow. In the case of a misuser exploiting the vulnerability by accident, are the odds good that this attacker will stumble on the circumstances required to damage the application?

By taking the average of each of the DREAD scores for a vulnerability, you will arrive at a final number representing the “importance” of that vulnerability. It’s taken a long time to get here, from profiling the attackers through identifying how they might misuse the app and to what end, to discovering how the application lets them in to perform their attacks. But it’s been worth it: by choosing to address the most important vulnerabilities first you can be confident that time spent in securing your app is time well spent.

SUMMARY

Users of your application will have concerns about being able to complete their tasks with the application without being exposed to unacceptable levels of risk. In constructing a threat model of the application, you discover what the risks are, which risks your users will be most concerned about, and whether it is appropriate to accept or mitigate those risks.

These risks exist because your app can potentially be misused by an attacker — the potential misuse being termed a threat. The application contains, or gives access to, assets that are valuable to the attacker, user, or both. Attackers threaten the application to damage, destroy, or gain access to one or more assets. To do so, attackers must use one or more entry points — interfaces between the application and the outside world. Understanding how each entry point can gain an attacker access to the application’s assets is the key to assessing — and ultimately controlling — the security risk.

A vulnerability exists in your application if a threat can be realized by an attacker. You can class vulnerabilities using the DREAD system, rating the importance of the vulnerabilities by measuring various properties related to the likelihood and to the damage possible were any vulnerability to be exploited by an attacker.

While the users' worries and the threats they face in using the application do not relate to any specific technology, the details of a vulnerability and the most appropriate way to mitigate it in a Cocoa or Cocoa Touch application are strongly dependent on the Objective-C language and the features and APIs available in Apple's frameworks. The remaining chapters of this book describe how the various features of the Mac and iPhone platforms are relevant to the security of an application running on those platforms, the problems those features can introduce, and how you can use them to mitigate the risks faced by your users.

