Chapter 21. Secure Application Architecture

The first step in securing any web application is the architecture phase. When building a product, a cross-functional team of software engineers and product managers usually collaborates to find a technical model that will serve a very specific business goal in an efficient manner. In software engineering, the role of an architect is to design modules at a high level and evaluate the best ways for modules to communicate with each other. This can be extended to determining the best ways to store data, what third-party dependencies to rely on, what programming paradigm should be predominant throughout the codebase, etc.

Similarly to a building architect, software architecture is a delicate process that carries a large amount of risk because re-architecture and refactor are expensive processes once an application has already been built. Security architecture includes a similar risk profile to software or building architecture. Often, vulnerabilities can be prevented easily in the architecture phase with careful planning and evaluation. However, too little planning, and application code must be re-architected and refactored—often at a large cost to the business.

The NIST has claimed, based on a study of popular web applications, that "The cost of removing an application security vulnerability during the design phase ranges from 30–60 times less than if removed during production." Hence solidifying any doubts we have regarding the importance of the architecture phase.

Analyzing Feature Requirements

The first step in ensuring that a product or feature is architected securely is collecting all of the business requirements that the product or feature is expected to implement. Business requirements can be evaluated for risk prior to their integration in a web application even being considered.

TIP

Any organization that has separate teams for security and R&D should ensure that communication pathways between the two are built into the development process. Features cannot be properly analyzed in a silo, and such analysis should include stakeholders from engineering as well as product development.

Consider this business case: after cleaning up multiple security holes in its codebase, MegaBank has decided to capitalize on its newly found popularity by beginning its own merchandising brand. MegaBank's new merchandising brand, MegaMerch, will offer a collection of high-quality cotton T-shirts, comfortable cotton/elastic sweatpants, and men's and women's swimwear with the MegaMerch (MM) logo.

In order to distribute merchandise under the new MegaMerch brand, MegaBank would like to set up an ecommerce application that meets the following requirements:

- Users can create accounts and sign in.
- User accounts contain the user's full name, address, and date of birth.
- Users can access the front page of the store that shows items.
- Users can search for specific items.
- Users can save credit cards and bank accounts for later use.

A high-level analysis of these requirements tells us a few important tidbits of information:

- We are storing credentials.

- We are storing personal identifier information.
- Users have elevated privileges compared to guests.
- Users can search through existing items.
- We are storing financial data.

These points, while not out of the ordinary, allow us to derive an initial analysis of what potential risks this application could encounter if not architected correctly. A few of the risk areas derived from this analysis are as follows:

Authentication and authorization

How do we handle sessions, logins, and cookies?

Personal data

Is it handled differently than other data? Do laws affect how we should handle this data?

Search engine

How is the search engine implemented? Does it draw from the primary database as its single source of truth or use a separate cached database?

Each of these risks brings up many questions about implementation details, which provide surface area for a security engineer to assist in developing the application in a more secure direction.

Authentication and Authorization

Because we are storing credentials and offering a different user experience to guests and registered users, we know we have both an authentication and an authorization system. This means we must allow users to log in, as well as be able to differentiate among different tiers of users when determining what actions these users are allowed.

Furthermore, because we are storing credentials and supporting a login flow, we know there are going to be credentials sent over the network.

These credentials must also be stored in a database, otherwise the authentication flow will break down.

This means we have to consider the following risks:

- How do we handle data in transit?
- How do we handle the storage of credentials?
- How do we handle various authorization levels of users?

Secure Sockets Layer and Transport Layer Security

One of the most important architectural decisions to tackle as a result of the risks we have identified is how to handle data in transit. Data in transit is an important first-step evaluation during architecture review because it will affect the flow of all data throughout the web application.

An initial data-in-transit requirement should be that all data sent over the network is encrypted en route. This reduces the risk of a man-in-the-middle attack, which could steal credentials from our users and make purchases on their behalf (since we are storing their financial data).

Secure Sockets Layer (SSL) and Transport Layer Security (TLS) are the two major cryptographic protocols in use today for securing in-transit data from malicious eyes in the middle of any network. SSL was designed by Netscape in the mid-1990s, and several versions of the protocol have been released since then.

TLS was defined by RFC 2246 in 1999 and offered upgraded security in response to several architectural issues in SSL. TLS cannot interpolate with older versions of SSL due to the amount of architectural differences between the two. TLS offers the most rigid security, whereas SSL has higher adoption but multiple vulnerabilities that reduce its integrity as a cryptographic protocol. See Figure 21-1 for an example of a method of implementing TLS.

Figure 21-1. Let's Encrypt is one of only a few nonprofit security authorities (SA) that provide certificates for TLS encryption

All major web browsers today will show a lock icon in the URL address bar when a website's communication is properly secured via SSL or TLS. The HTTP specification offers "HTTPS" or "HTTP Secure," a Uniform Resource Identifier (URI) scheme that requires TLS/SSL to be present before allowing any data to be sent over the network. Browsers that support HTTPS will display a warning to the end user if TLS/SSL connections are compromised when an HTTPS request is made.

For MegaMerch, we would want to ensure that all data is encrypted and TLS compatible prior to being sent over the network. The way TLS is implemented is generally server specific, but every major web server software package offers an easy integration to begin encrypting web traffic.

Secure Credentials

Password security requirements exist for a number of reasons, but unfortunately, most developers don't understand what makes a password hacker-safe. Creating a secure password has less to do with the length

and number of special characters, but instead has everything to do with the patterns that can be found in the password. In cryptography, this is known as *entropy*—the amount of randomness and uncertainty. You want passwords with a lot of entropy.

Believe it or not, most passwords used on the web are not unique. When a hacker attempts to brute force logins to a web application, the easiest route is to find a list of the top most common passwords and use that to perform a dictionary attack. An advanced dictionary attack will also include combinations of common passwords, common password structure, and common combinations of passwords. Beyond that, classical brute forcing involves iterating through all possible combinations.

As you can see, it is not so much the length of the password that will protect you, but instead the lack of observable patterns and avoidance of common words and phrases. Unfortunately, it is difficult to convey this to users. Instead, we should make it difficult for a user to develop a password that contains a number of well-known patterns by having certain requirements.

For example, we can reject any password in a top one thousand password list and tell the user it is too common. We should also prevent our users from using birthdates, first name, last name, or any part of their address. At MegaMerch, we can require first name, last name, and birthdate at signup and prevent these from being allowed within the user's password.

Hashing Credentials

When storing sensitive credentials, we should never store in plain text. Instead, we should hash the password the first time we see it prior to storing it. Hashing a password is not a difficult process, and the security benefits are massive.

Hashing algorithms differ from most encryption algorithms for a number of reasons. First off, hashing algorithms are not reversible. This is a key point when dealing with passwords. We don't want even our own staff to be able to steal user passwords because they might use those passwords

elsewhere (a bad practice, but common), and we don't want that type of liability in the case of a rogue employee.

Next, modern hashing algorithms are extremely efficient. Today's hashing algorithms can represent multiple-megabyte strings of characters in just 128 to 264 bits of data. This means that when we do a password check, we will rehash the user's password at login and compare it to the hashed password in the database. Even if the user has a huge password, we will be able to perform the lookup at high speeds.

Another key advantage of using a hash is that modern hashing algorithms have almost no collision in practical application (either 0 collisions or statistically approaching 0—1/1,000,000,000+). This means you can mathematically determine that the probability that two passwords will have identical hashes will be extraordinarily low. As a result, you do not need to worry about hackers "guessing" a password unless they guess the exact password of another user.

If a database is breached and data is stolen, properly hashed passwords protect your users. The hacker will only have access to the hash, and it will be very unlikely that even a single password in your database will be reverse engineered.

Let's consider three cases where a hacker gets access to MegaMerch's databases:

Case #1

Passwords stored in plain text

Result

All passwords compromised

Case #2

Passwords hashed with MD5 algorithm

Result

Hacker can crack some of the passwords using rainbow tables (a precomputed table of hash→password; weaker hashing algorithms are susceptible to these)

Case #3

Passwords hashed with BCrypt

Result

It is unlikely any passwords will be cracked

As you can see, all passwords should be hashed. Furthermore, the algorithm used for hashing should be evaluated based on its mathematical integrity and scalability with modern hardware. Algorithms should be SLOW on modern hardware when hashing, hence reducing the number of guesses per second a hacker can make.

When cracking passwords, slow hashing algorithms are essential because the hacker will be automating the password-to-hash process. Once the hacker finds an identical hash to a password (ignoring potential collision), the password has been effectively breached. Extremely slow-to-hash algorithms like BCrypt can take years or more to crack one password on modern hardware.

Modern web applications should consider the following hashing algorithms for securing the integrity of their users' credentials.

BCrypt

BCrypt is a hashing function that derives its name from two developments: the "B" comes from Blowfish Cipher, a symmetric-key block cipher developed in 1993 by Bruce Schneier, designed as a general purpose and open source encryption algorithm. "Crypt" is the name of the default hashing function that shipped with Unix OSs.

The Crypt hashing function was written with early Unix hardware in mind, which meant that at the time hardware could not hash enough passwords per second to reverse engineer a hashed password using the

Crypt function. At the time of its development, Crypt could hash fewer than 10 passwords per second. With modern hardware, the Crypt function can be used to hash tens of thousands of passwords per second. This makes breaking a Crypt-hashed password an easy operation for any current-era hacker.

BCrypt iterates on both Blowfish and Crypt by offering a hashing algorithm that actually becomes slower on faster hardware. BCrypt-hashed passwords scale into the future because the more powerful the hardware attempting to hash using BCrypt, the more operations are required. As a result, it is nearly impossible for a hacker today to write a script that would perform enough hashes to match a complex password using brute force.

PBKDF2

As an alternative to BCrypt, the PBKDF2 hashing algorithm can also be used to secure passwords. PBKDF2 is based on a concept known as *key stretching*. Key stretching algorithms will rapidly generate a hash on the first attempt, but each additional attempt will become slower and slower. As a result, PBKDF2 makes brute forcing a computationally expensive process. PBKDF2 was not originally designed for hashing passwords but should be sufficient for hashing passwords when BCrypt-like algorithms are not available.

PBKDF2 takes a configuration option that represents the minimum number of iterations in order to generate a hash. This minimum should always be set to the highest number of iterations your hardware can handle. You never know what type of hardware a hacker might have access to, so by setting the minimum iterations for a hash to your hardware's maximum value, you are eliminating potential iterations on faster hardware and eliminating any attempts on slower hardware.

In our evaluation of MegaMerch, we have decided to hash our passwords using BCrypt and will only compare password hashes.

MFA

In addition to requiring secure, hashed passwords that are encrypted in transit, we also should consider offering multifactor authentication (MFA) to our users who want to ensure their account integrity is not compromised. Figure 21-2 shows Google Authenticator, one of the most common MFA applications for Android and iOS. It is compatible with many websites and has an open API for integrating into your application. MFA is a fantastic security feature that operates very effectively based on a very simple principle.

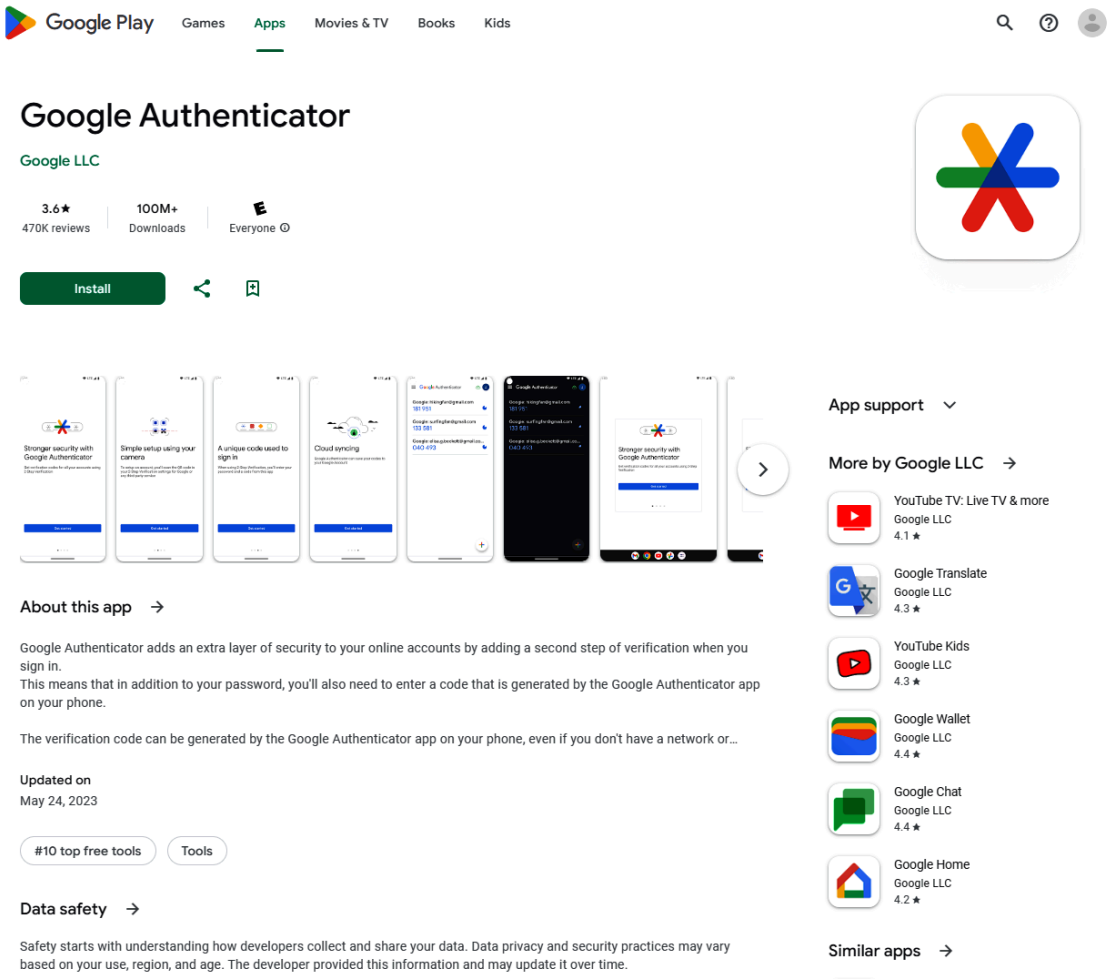

Figure 21-2. Google Authenticator—one of the most commonly used MFA applications for Android and iOS

Most MFA systems require a user to enter a password into their browser, in addition to entering a password generated from a mobile application or SMS text message. More advanced MFA protocols actually make use of a physical hardware token, usually a USB drive that generates a unique one-time-use token when plugged into a user's computer. Generally speaking, the physical tokens are more applicable to the employees of a

business than to its users. Distributing and managing physical tokens for an ecommerce platform would be a painful experience for everyone involved. Phone app/SMS-based MFA might not be as secure as a dedicated MFA USB token, but the benefits are still an order of magnitude safer than application use without MFA.

In the absence of any vulnerabilities in the MFA app or messaging protocol, MFA eliminates remote logins to your web application that were not initiated by the owner of the account. The only way to compromise an MFA account is to gain access to both the account password and the physical device containing the MFA codes (usually a phone).

During our architecture review with MegaMerch, we strongly suggest offering MFA to users who wish to improve the security of their MegaMerch accounts.

PII and Financial Data

When we store personally identifiable information (PII) on a user, we need to ensure that such storage is legal in the countries we are operating in, and that we are following any applicable laws for PII storage in those countries. Beyond that, we want to ensure that in the case of a database breach or server compromise, the PII is not exposed in a format that makes it easily abusable. Similar rules to PII apply to financial data, such as credit card numbers (also included under PII laws in some countries).

A smaller company might find that rather than storing PII and financial details on its own, a more effective strategy could be to outsource the storage of such data to a compliant business that specializes in data storage of that type.

Search Engines

Any web application implementing its own custom search engine should consider the implications of such a task. Search engines typically require data to be stored in a way that makes particular queries very efficient.

How data is ideally stored in a search engine is much different than how data is ideally stored in a general purpose database.

As a result, most web applications implementing a search engine will need a separate database from which the search engine draws its data. As you can clearly see, this could cause a number of complications, requiring proper security architecture up front rather than later.

Syncing any two databases is a big undertaking. If the permissions model in the primary database is updated, the search engine's database must be updated to reflect the changes in the primary database.

Additionally, it's possible that bugs introduced into the codebase might cause certain models to be deleted in the primary database but not in the search database. Alternatively, metadata in the search database regarding a particular object may still be searchable after the object has been removed from the primary database.

All of these are examples of concerns when implementing search that should definitely be considered before implementing any search engine, be it Elasticsearch or an in-house solution. Elasticsearch is the largest and most extensively used open source distributed search (Figure 21-3). It's easily configurable, well documented, and can be used in any application free of charge. It is based on top of Apache's Solr search engine project.

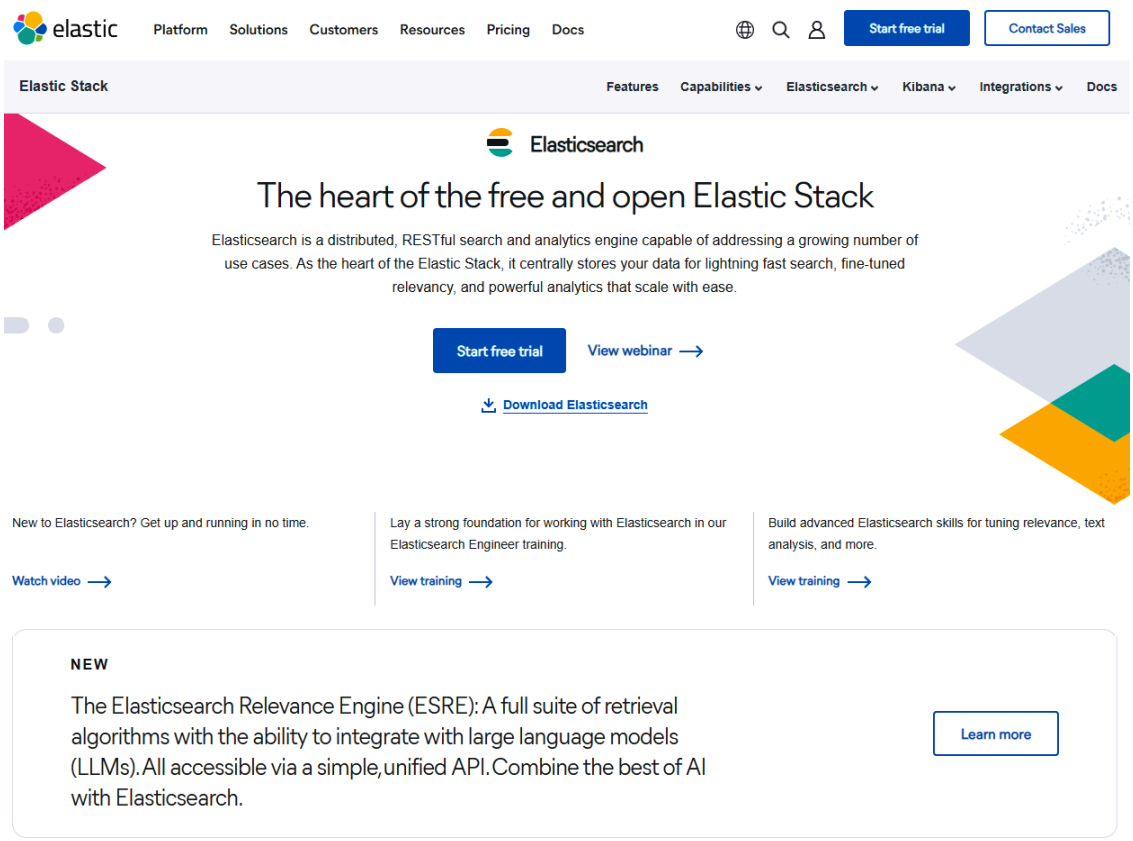

Figure 21-3. The Elasticsearch search engine

Zero Trust Architecture

Zero Trust Architecture is the application of a philosophy called *zero trust* to software architecture. The two terms are used interchangeably here. In general, Zero Trust Architecture is known by a number of different names:

- Zero Trust
- Zero Trust Network Access (ZTNA)
- Zero Trust Design
- Zero Trust Pattern
- Zero Trust Design Pattern

All of these terms refer to the same set of design principles, which we discuss in this section.

The History of Zero Trust

The term *zero trust* first appeared in computer scientist Stephen Paul Marsh's 1994 doctoral thesis at the University of Stirling (Scotland).

Stephen argued that the concept of *trust* could be broken down into a finite set of criteria that a computer could compute. Shortly after the publication of Stephen's doctoral thesis, the term "zero trust" disappeared from common vernacular until 2018.

In 2020, the NIST published a whitepaper dubbed *Zero Trust Architecture (SP-800-207)*. This publication revitalized the use of the term "zero trust," which became one of the most widely searched information security terms for years after its publication. Most of the time today, *zero trust* refers to the guidelines written in the 2020 NIST whitepaper of the same name when used in a security setting.

Implicit Versus Explicit Trust

At its core, Zero Trust Architecture is a design pattern for developing secure applications. Zero trust primarily distinguishes itself from other design patterns by putting a focus on two primary types of "trust": *implicit* and *explicit* trust. In the aforementioned NIST whitepaper, implicit trust is defined as the type of trust that is granted based on proximity or roles.

Consider a castle with a moat surrounding it. Because the purpose of the moat is to keep unintended visitors out of the castle, it could be *assumed* that anyone within the castle who has already passed the moat boundary should be trusted. This is an implicit trust model because there is no *verification* beyond the moat. In this example, the moat is the sole determiner if an individual should be trusted or untrusted—leaving no consideration to an edge case where an unwanted visitor swims across the moat.

A similar and more relevant example of implicit trust might be a network that is protected by a firewall but has no verification for any requests that have made it past the firewall. The NIST argues that this type of trust model is outdated and leads to the production of vulnerable and easily exploitable web applications.

A better model, per the NIST's guidance, is the *trust but verify* or *explicit trust* model. In such a model, applications are designed so that verification occurs whenever a privileged functionality is invoked. This means

that simply existing in the implicit trust zone (e.g., AWS cloud account, passing server firewall, previous successful authentication check, etc.) is not sufficient.

According to Zero Trust Architecture, every action that could lead to a compromise must have a verification step in front of it to ensure that the requester or invoker is who they claim to be and still holds proper permissions.

Authentication and Authorization

You have probably noted so far that Zero Trust Architecture has a lot of overlap with *the principle of least privilege*. This is the case because at its core, Zero Trust Architecture spends a lot of time ensuring that an actor (script, user, etc.) is only able to access the resources that are intended for them.

But there are other cases to consider where zero trust is beneficial as well. Consider the case where an employee at a business holds a high-ranking role, and with that role comes significant permissions within the company's internal software. We will call this employee Joe Admin since the employee has admin permissions that enable high levels of access to internal software.

Throughout his day-to-day work, Joe makes use of a multitude of software services to view company financials and evaluate business deals, among other functions. He even has access to company bank accounts.

One day, Joe is fired from his company for money laundering (or any other legitimate reason). The internal software at this company was architected such that tokens used for authentication and authorization have a 48-hour expiration window.

This is not uncommon, and in this case the software Joe has been using just looks for a valid authorization token and does not verify in any way that Joe's status as a high-ranking employee is unchanged. Because of this,

Joe is able to utilize said software even after being fired and cause additional chaos by modifying the company books in an attempt for revenge.

Zero Trust Authorization, or authorization architecture that applies zero trust principles, would have prevented this edge case from occurring. The attack occurred because trust was granted implicitly to any user with an unexpired access token—without considering a possible change of user employment state. Implementing NIST-defined zero trust into each authorization step would have resulted in continuous authorization, which would have prevented this type of attack against the internal system.

Summary

There are many concerns to consider when building an application. Whenever a new application is being developed by a product organization, the design and architecture of the application should also be analyzed carefully by a skilled security engineer or architect. Deep security flaws—such as an improper authentication scheme or half-baked integration with a search engine—could expose your application to risk that is not easily resolved. Once paying customers begin relying on your application in their workflows, especially after contracts are written and signed, resolving architecture-level security bugs will become a daunting task.

At the beginning of this chapter, I included the estimate from NIST that a security flaw found in the architecture phase of an application could cost 30 to 60 times less to fix than if it is found in production. This can be because of a combination of factors, including the following:

- Customers may be relying on insecure functionality, hence causing you to build secure equivalent functionality and provide them with a migration plan so that downtime is not encountered.
- Deep architecture-level security flaws may require rewriting a significant number of modules in addition to the insecure module. For example, a complex 3D video game with a flawed multiplayer module may require rewriting not only the networking module but the game modules written on top of the multiplayer networking module as

well. This is especially true if an underlying technology has to be swapped out to improve security (moving from User Datagram Protocol or TCP networking, for example).
- The security flaw may have been exploited, costing the business actual money in addition to engineering time.
- The security flaw may be published, bringing bad PR against the affected web application, costing the business in lost engagements and customers who will choose to leave.

Ultimately, the ideal phase to catch and resolve security concerns is always the architecture phase. Eliminating security issues in this phase will save you money in the long run and eliminate potential headaches caused by external discovery or publication later on.