Chapter 23. Secure User Experience

An often forgotten component of a secure web application is that of the user interface. UIs are the standard way for an end user to interact with any web application. Highly specialized web applications may allow a user to interact via a CLI, a REST API, or even via an XML or JSON file. But the majority of applications and end users prefer to interact via user interface.

On the web, UIs are constrained to a small set of technologies due to browser computing model limitations. Typically these interfaces are HTML, CSS, and JavaScript, but may make use of plug-ins or applets (e.g., Java) despite security and performance limitations.

In this chapter, we won't focus on the technologies that power a UI (e.g., JavaScript, CSS, HTML). Instead we will focus on the design, experience, and useability of the application output by these technologies, which is then interacted with via the end user. Without further ado, let's consider some common security mistakes that developers make when producing UIs, and then evaluate potential solutions for those security gaps.

Information Disclosures and Enumeration

Both information disclosure and enumeration vulnerabilities share a common set of identifying features, and (often) stem from the need to power a UI with useful data. However, both of these vulnerabilities can be avoided or at least mitigated with smart user experience design.

Information Disclosures

The first and most important thing to consider when designing a UI for a web application is how much information you intend to give to an end user. At first glance this may seem like an easy problem to solve, but with further consideration, you may see that the primary purpose of a UI is to provide information to an end user. In fact, to the end user, an interface is often better if it provides more information rather than less.

From this perspective, we are then left with the challenge of determining how much information should be provided to the user—and what information should be kept hidden from the end user. This comes with the understanding that there may be trade-offs where the end user's experience is degraded in exchange for better information security.

There are several classic, and simple, examples of where additional information disclosure can produce a security risk to your web application—despite being done with the intent of improving a user's experience. Let's consider a case in which you are a security engineer working on a web application alongside several talented UX designers.

The designers on your team (rightfully so) wish to improve the user experience by providing detailed error messages to an end user so the end user can be aware of what type of issue occurred when a form in the web application fails form submission. There are many ways such an error message can be implemented. Let's consider the following possible implementations:

- Reflect the server's error message and error code back to the user
- Reflect the server's error message and error code back to the user, provided the message is found in a predefined allowlist
- Provide a general error message that is determined in the client code, alongside the server's error code
- Provide only the server error code
- Provide neither an error message nor the server's error code

Interestingly, it turns out that providing neither an error message nor the server's error code is (technically speaking) the most secure option—al-

though it is probably not the correct option due to the usability trade-offs.

This is a case where trade-off evaluation matters. Developers should never disclose information regarding the application's source code that could allow fingerprinting, but information regarding server state may be valuable to disclose to the user.

When disclosing information regarding server state to the end user, it's important to ensure it's filtered and formatted in a way that minimizes the potential risk. For example, rather than reflecting an error message directly from the database or middleware when a query fails, create an allowlist of common failure cases in a generic tone and return the closest match.

Consider the following error message: "the user Jonathan Smith could not be queried because the address 905 N. Main St. provided does not match the current file."

This error message could be replaced with a generic message that does not leak the address and hence reduces the chances of an accidental information disclosure. An improved generic error message could look like "The user could not be queried due to an address mismatch."

While still providing more information to the client than simply returning "an error occurred," the allowlisted generic error approach provides the information in regard to the cause without significant risk of information disclosure.

Remember, server-side error messages change over time as databases and middleware upgrade. So without a robust allowlist of predetermined error messages, it is possible information disclosure may not occur now but could occur at a later point in time.

When handing back HTTP error codes to the client alongside error messages, the best practice is often to use a generic code so that recon cannot be easily performed against your web server. The generic error code "400 - bad request" is often chosen for this because it provides less data than more robust and specialized error codes.

However, if you have a sense of humor you may choose to send back “418 - I’m a teapot,” which is an unused error code that was included in the HTTP specification back in 1998 as an April Fools’ joke (but is still supported by most web servers). The benefit of using an error code like “418 - I’m a teapot” is because it has no official real-world use, it is highly unlikely to interfere with any third-party tools used by your developers.

There are cases where nongeneric error codes may be preferential, for example, if your API is consumed by multiple clients that expect detailed information about the status of a network call. In this case, offering more specialized error codes is acceptable, but each API endpoint should be evaluated on a case-by-case basis to determine if the error codes provided could leak any form of application state that could be exploited.

Enumeration

Enumeration vulnerabilities occur when *multiple* queries with a similar structure result in an information disclosure, or when the sum of those queries (or combination of queries) can be used to deduce information that the client is not intended to have access to. There are many examples of enumeration vulnerabilities leading to information being disclosed in an unintended way.

For example, if an API `GET /user/:id` permits the lookup of a single user, and the user’s `:id` column in the database is an iterable integer (e.g., 1, 2, 3, etc.), than by repeatedly requesting the `GET /user/:id` endpoint with successive numbers, an attacker would be able to deduce the total number of users within a system. This is likely not high-risk information, but it is still information that was not intended to be disclosed.

A slightly more severe example of an enumeration vulnerability is as follows:

1. A hacker has found an admin interface that is not intended to be exposed to the internet. This admin interface is for MegaBank and allows a specific subset of MegaBank users to remotely administer user accounts and balances.

2. The hacker cannot breach the authentication mechanisms and, as a result, opts to try to find a valid username:password combination in order to log in as an employee.
3. The hacker notices that some username:password authentication attempts result in the error “wrong password,” while others result in the error “user does not exist.”
4. The hacker collects a list of data breaches for other websites, with the assumption that some of MegaBank’s staff would have an account on these breached websites. The breach data contains usernames and email addresses. Example breached websites include hobby forums and small ecommerce websites.
5. The hacker finds *@megabank.com* staff email addresses and plugs the associated usernames into the admin API, generating a spreadsheet of users that return the error “wrong password” during an auth attempt (this is the *enumeration* step).
6. Now that the hacker has determined which MegaBank usernames work on the admin account, the hacker can make use of other techniques (data breaches, social engineering, brute force, dictionaries) to find a valid password and log in.

You may be asking why this form of enumeration is important. Couldn’t the attacker have simply brute forced the combinations from the start?

The truth of the matter is that brute force, dictionary attacks, and other methods of compromising a user account lack accuracy, are easily detectable, and sometimes are not mathematically feasible. By finding and exploiting an enumeration bug, the complexity of a brute force attack is significantly reduced because username:password combinations are no longer the goal but rather just a valid password for a known username.

Furthermore, if detection is a risk or if rate limits are an issue, an attacker can make use of highly targeted methods of collecting a password (e.g., spear phishing), and these methods would go undetected.

The most common mechanism for avoiding enumeration in your application is to design it in such a way that the following three criteria are met:

- When errors are provided to an end user, they are provided in a generic way (similar to mitigating information disclosures) so multiple calls cannot amass data that can be used maliciously.
  For example, OWASP suggests all authentication failures use the generic error message “authentication failed” rather than specifying if the username or password (or combination) led to the failure.
- When designing any form of data structure or code module that can be interacted with via a user, any easily guessable pattern or series of characters should be avoided.
  For example, naming your API endpoints 1/2/3/4/5 would be considered a bad practice. Likewise, creating user IDs based on sign-up date would be considered a bad practice.
- Any and all API endpoints that could potentially be at risk of enumeration enforce rate limits to reduce the risk of programmatic repeated access exposing valuable information to an attacker.
  Rate limits should be determined by looking at the average number of requests a legitimate user needs access to, and creating a limit at the upper bound of the legitimate use model.

In conclusion, enumeration vulnerabilities are mitigated in many of the same ways as information disclosure vulnerabilities. If you consider enumeration to be a form of information disclosure that occurs when small bits of relatively harmless data are leaked over and over, mitigating such attacks becomes quite simple.

Secure User Experience Best Practices

In addition to vulnerabilities and security risks that present themselves at the user experience layer, the user experience layer in an application is one of the ideal places to educate and guide your end users toward using your application in a secure manner.

Wikipedia defines a *dark pattern* as a UI design pattern that tricks users into invoking functionality the user does not intend to invoke. This often leads to a degradation of security or privacy for the user.

The lesser known flip-side of a dark pattern is what I call a *light pattern*, a pattern that gently guides users to improving their security and privacy (Figure 23-1). Light patterns can be beneficial because users often don't understand enough about how a web application works to make the best security choices for themselves.

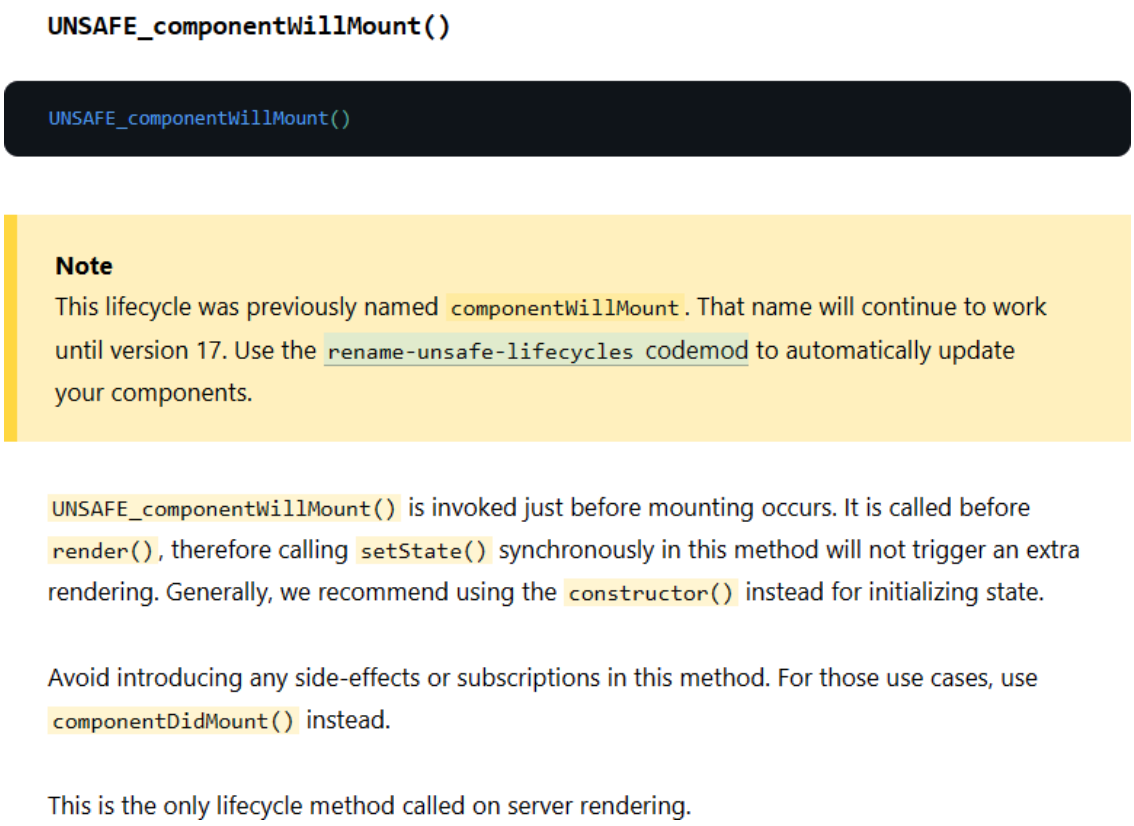

Figure 23-1. The popular JavaScript library React uses light patterns in its naming conventions to gently guide developers away from accidentally making use of functions that contain security risk

A link to a documentation page next to a security configuration setting in a web UI is not a light pattern. It is documentation. With any substantial sample of users for any given web application of reasonable complexity, we can assume that only a subset of users will read documentation in its entirety. This is where light patterns become useful. Rather than prompting the user to understand a security mechanism that is optional via adding documentation to the page where the security mechanism is enabled or disabled, light patterns can be scattered throughout an application whenever a feature containing security or privacy risks is invoked.

Take, for example, a web-based cryptocurrency wallet that enables transfers from one user to another. By default, most wallets allow outbound transfers of any amount. But many web-based wallets offer opt-in restric-

tion configurations that can limit the blast radius of a compromised account.

An example of a security mechanism would be permitting only $1,000 worth of currency outflows per day in a wallet containing $100,000 of total cryptocurrency. By limiting the daily outflow via a daily cap, even if the wallet is compromised, it would take an attacker one hundred days to liquidate the user's entire savings. This allows the user time to access their wallet and eliminate the attacker's access.

Such mechanisms are crucial to a user's security but rarely enabled. By adding light patterns to the application's UI, the conversion rate of users who enable the aforementioned security feature could be dramatically increased. But where and how should a light pattern with this goal be implemented?

The best way to add such a light pattern is, in fact, not on the settings page where the security control is implemented, but instead across all of the functionality where the security risk is invoked. On every outgoing transaction, the user should be warned that the security setting is not enabled and mistyping the wrong amount could result in irreversible financial damages. In this way, the user is not burdened with considering an attacker but instead provided with an edge case where currency could be lost based on their own accidental misuse.

Providing this scenario to the user on every transaction will convert better because the user is, in fact, sending currency and has direct familiarity with the possibility of such an accident occurring. This is more effective than using the case where an attacker gains access to their account because the user may not have a metric for how probable an account takeover is.

Gently providing the user with an understanding of application risks and security mechanisms whenever a risky action is invoked guides the user to create a connection between action and consequence. This pattern will often lead the user to enable a security mechanism at a much higher rate than simply providing the user with documentation that lacks a connection and will, in most cases, not even be read in its entirety.

Note that secure by default is still a superior design pattern to light patterns, but light patterns should be used in cases where secure by default (e.g., force user to enable security mechanism by default) is not feasible.

Summary

To conclude, while the majority of vulnerabilities you will find arise from your application code, you should also always be aware of the ways in which you are designing your user experience. Because the primary goal of a UI is to present data to an end user, it is imperative to be aware of the ways in which your data is being presented so that the wrong data is not accessible.

Finally, a good user experience can be used to guide end users toward employing an application in a more secure manner. For functionality-rich applications where abuse potential exists, make sure to identify and alert your users whenever they are at risk—leading them down the most secure path available so that they can avoid common pitfalls that might result from a poor user experience.