Secure Web Application Development

Beginning web application developers have a tendency to focus so strongly on getting an application to work correctly that they forget one critical component of development—security. To effectively protect a web application, developers have to think like hackers and have to know what kinds of attacks to expect, which is difficult for beginning developers who lack experience with hackers. Hackers can exploit security vulnerabilities to access or delete user data, to break the application's functionality, to prevent legitimate users from accessing the application, or even to gain control of the servers the application is running on.

Security is a big issue in the professional world, and even professionals make mistakes. Consider these security breaches that made headlines:

- In December 2013, hackers accessed credit card data for 40 million Target shoppers, leading to over 90 lawsuits and a noticeable decline in Target's sales (Riley, Elgin, Lawrence, & Matlack, 2014).
- In an attack on Home Depot's systems nearly a year later, 56 million credit card numbers and email addresses were compromised (Banjo, 2014).
- A well-orchestrated attack on Code Spaces, an application where users could collaboratively write source code, forced the company out of business when the attackers wiped its databases (Code Spaces, 2014).

Developers must understand how to secure their web applications if they want to protect their users, data, and servers. This chapter discusses common attacks that hackers use and what developers can do to defend against them.

What is a Web Application?

Before learning about web application security, it is important to know exactly what defines a web application. A web application is a program that runs on computers called *servers* that the application developer either owns or rents. Those who use web applications interact with them from computer programs called *clients*. There may be multiple clients, each with a different interface, for a single web application. For example, Twitter is a single application that can be run in a desktop browser, a mobile browser, an iPhone app, or an Android app. Each Twitter client has a slightly different user interface, but each interacts with the same Twitter application by connecting to Twitter's servers. Clients interact with servers by making requests to get, create, update, or delete data. Servers listen to the clients' requests, run the application to fulfill those requests, and then send appropriate responses to the clients.

Protecting Web Applications from Common Attacks

Beginning developers frequently do not know how to defend their web applications, because they are unfamiliar with methods that hackers use to attack. Hackers use a variety of attacks in an effort to access sensitive user data. If developers want to know how to effectively prevent attacks to keep data secure, they must be familiar with attacks that hackers might use against their applications. Common attacks include (but are not limited to) packet sniffing, bypassing authorization rules, password cracking, code injection, distributed denial-of service attacks, and buffer overflow attacks. The following sections will describe how these attacks work and how developers can defend their applications from these attacks.

Packet Sniffing

The internet works by sending electromagnetic signals through wires, fiber optic cables, and the air. These signals represent the data sent between machines and are separated into small groups of data

known as *packets*. Packet sniffing occurs when the attacker uses a wire tap or a radio receiver to record packets that are in transit. All packets sent over the internet are vulnerable to packet sniffing. The only way to protect sensitive data is to encode the data in such a way that only the intended recipient can decode it. In other words, data must be encrypted while in transit in order to be secure.

The HTTPS protocol. Fortunately, application developers are not expected to write code to encrypt data. There is an existing internet protocol that performs encryption and decryption operations: HTTPS. HTTPS uses a secure method (known as a handshake) to establish encryption and decryption keys between the server and the client. By enabling the HTTPS protocol on an application's web servers, the developer can ensure that all web traffic between the server and the client will be encrypted. If any packets are recorded by packet sniffers, the encrypted data will be indecipherable. The process of enabling HTTPS is different for every server, so developers will need to refer to their server's documentation for specific instructions. Though enabling HTTPS may seem like a hassle for a beginning developer, it is necessary to secure any sensitive information that flows between users and the application servers.

Email. If a web application communicates with users by sending emails, the developer must be careful to not include any sensitive information in those emails. Most email protocols (e.g., POP3, IMAP, SMTP, HTTP) do not encrypt data that is in transit. As a general rule, expect email to not be secure (Duncan, 2013). If a web application needs to communicate sensitive information to a user, the application should send a generic email prompting the user to log into the application and then display the sensitive information to the user once the user logs in.

Bypassing Authorization Rules

The process of logging into an application is also known as *authentication*. A user provides authentication tokens—typically a username and password—to the application, and the application

confirms that the tokens are authentic. Once a user's identity is confirmed, the application uses authorization rules to decide what data the user is allowed to access and what actions the user is allowed to perform. For example, a standard user may be authorized to write comments, view personal comments, and view comments written by friends. An administrative user, on the other hand, may be authorized to view, modify, or delete any comments.

A common mistake that beginning developers make is to use client-side code to enforce authorization rules. Client-side code is code that is executed on the user's device. For a web application, client-side code is typically HTML and JavaScript, and it is executed in the user's web browser. In the previous example, the HTML for an administrative user may include a delete button to delete comments, though the delete button would be hidden for standard users. Displaying only HTML elements that the user is authorized to use is a good design practice, but it is not sufficient for enforcing authorization rules. Because client-side code is executed on the user's device, there is no guarantee that the client will execute the code in the way the developer intended. Even if a standard user cannot see the delete button, it is still possible for the user to bypass the authorization rules built into the client-side code and submit a delete request directly to the application's servers. Thus, authorization rules cannot be enforced in client-side code alone but must always be enforced in server-side code.

Server-side code is code that is executed on the application's servers. Common server-side programming languages include Java, ASP.NET, PHP, Python, Ruby, and Node.js. Server-side code acts as an intermediary between the client and the data, so data will not be secure if the server-side code fails to protect it. Whenever a user makes a request to one of the application's servers, the server should verify that the user is authorized to perform the request. If the user is not authorized, the server should respond with an error message and terminate the request. Then, even if a user is able to circumvent any client-side defenses, the server will enforce the authorization rules to help keep the data secure.

Password Cracking

If attackers cannot find a way to bypass authorization rules, they may try to guess or steal the authentication tokens of other users. Although users are primarily responsible for securing their own passwords, there are ways for application developers to help protect user passwords:

- Never send a password in an email or over a connection that is not using the HTTPS protocol. Only exchange passwords over encrypted connections to protect passwords from packet sniffing.
- Never store raw passwords. Instead, use a cryptographic hash function (such as a SHA-2 algorithm) on a password to get a digest, then store the digest. When a user provides a username and password, run the same hash function on the given password and compare that digest to the stored digest to authenticate. If an attacker manages to steal the application's password digests, it will still be very difficult for the attacker to recover the original passwords.
- Enforce a cap on how many times a user is allowed to attempt to log in with a wrong password. Attackers may try to use computer programs to submit thousands of authentication attempts in just a few seconds. If a single user fails to provide a correct password after several attempts, the application should not allow the user to try to authenticate again for a period of time. This prevents an attacker from trying millions of guesses until the correct password is found.

By helping users secure their passwords, application developers can reduce the likelihood of successful attacks and keep sensitive user data from getting into the wrong hands.

Code Injection

If a web application stores user input in a database or shares one user's input with other users, then the application is potentially vulnerable to an attack known as code injection. Code injection occurs when a user submits code in an input field, and the application unwittingly uses that input in such a way that any code in the input might be executed. The executed code could potentially allow the user to delete

the entire database, download sensitive data, or break the web application's functionality. To effectively protect web applications, developers must defend against two common code injection attacks: SQL injection and cross-site scripting.

SQL injection. SQL is a querying language that developers use to interact with relational databases. Hackers may try to use SQL injection to make unauthorized changes to a web application's database. Fig. 1 illustrates an example in which the developer uses PHP as the server-side programming language and SQL as the database language. The developer builds an insert statement (a line of SQL code used to add new data to a database) and places the user's comment into the insert statement. If the user submits a typical comment (e.g., "I had a great time at the lake today!"), then the comment will be successfully saved in the database. However, a hacker may try to attack the database by trying to submit SQL code as a user comment. For example, if the user submits "test`, `hacker123`, now()); DROP TABLE user_comments;--" as a comment, then all user comments will be deleted from the database, and the developer will be left with nothing but a group of angry users.

```
$username = $_SESSION['username']; // Get username for current user
$userComment = $_POST['userComment']; // Get user comment from browser input

$sqlStatement = "INSERT INTO user_comments
                (comment_text, author, created_date_time)
        VALUES  (`$userComment`, `$username`, now())";

/* If user comment is "I had a great time at the lake today!"
 * then the comment is successfully saved in the database */
$sqlStatement = "INSERT INTO user_comments
                (comment_text, author, created_date_time)
        VALUES  (`I had a great time at the lake today!`, `bob123`, now())";

/* If user comment is "test`, `hacker123`, now()); DROP TABLE user_comments;--"
 * then all user comments will be deleted */
$sqlStatement = "INSERT INTO user_comments
                (comment_text, author, created_date_time)
        VALUES  (`test`, `hacker123`, now()); DROP TABLE user_comments;--`, `bob123`, now())";
```

Figure 1. An example of how a hacker might use SQL injection to make unauthorized changes to data.

For a web application that needs to save user input into a relational database, the developer must protect against SQL injection to keep the data safe. Developers can use code libraries (such as PHP's PDO library) that can prepare SQL statements in a safe way by removing parts of user input that may be unsafe to execute (cf. Fig. 2). For databases that do not use SQL (e.g., MongoDB, Cassandra, Redis), there are still vulnerabilities to code injection attacks. To effectively defend web applications that use NoSQL databases, developers must research vulnerabilities and learn about recommended precautions to take.

```
$sqlStatement = $connection->prepare("INSERT INTO user_comments
                (comment_text, author, created_date_time)
        VALUES  (?, ?, now())"); // Use ? in place of input parameters
$input = array($userComment, $username);
$success = $sqlStatement->execute($input); // Sanitize and execute the query
```

Figure 2. An example of how a developer can use prepared statements to prevent SQL injection.

Cross-site scripting. Web applications that save input from one user and display it to other users are potentially vulnerable to cross-site scripting attacks, also known as XSS attacks or JavaScript injection. To attempt a cross-site scripting attack, a hacker submits HTML and JavaScript code as user input (e.g., "Hi everyone! <script> alert(document.cookie); </script> <iframe src='//hackers.com/unsafe_site.php'> </iframe>"). The hacker hopes that when the web application displays the comment to other users, the other users' browsers will render the HTML and execute the JavaScript. Hackers can potentially use cross-site scripting to steal sensitive data from other users, to open pop-ups or iframes to other websites, or to download malicious software onto other users' devices. There are two common methods used to protect web applications from cross-site scripting attacks: stripping HTML tags and whitelisting HTML tags.

Stripping HTML tags from user input is a relatively simple method to implement. A web application sanitizes user input by removing all HTML tags (defined by text between the < symbol and

the > symbol) from the input, then stores the sanitized input in the database. When the web application retrieves that data from the database and displays it to another user, the lack of HTML tags will allow the other user's client to display the input text without executing any code.

The main limitation of stripping away all HTML tags is that the developer may want to allow the user to input certain HTML tags—such as hyperlinks, lists, tables, or formatting tags—to make the application more flexible and user-friendly. Whitelisting HTML tags is done by enumerating a list of acceptable tags and removing all other tags from user input. This method may be difficult to implement for beginning developers, but many programming languages have libraries (such as the OWASP HTML sanitizer) that implement whitelisting functions that are relatively simple for developers to use. Due to the complexity of whitelisting functions, beginning developers may want to look for available sources before attempting to write these functions on their own.

Distributed Denial-of-Service

To execute a distributed denial-of-service (DDoS) attack, a hacker programs multiple devices (or *bots*) to flood a web application's servers with requests. The application servers utilize all of their resources to respond to the hacker's flood of requests, making it difficult for legitimate users to get any data from the application server. Servers may eventually be driven into a state of deadlock and crash. Because the hacker uses multiple bots with unique MAC addresses and IP addresses, it is difficult for the application servers to distinguish between the hacker's requests and legitimate requests. If a web application falls prey to a distributed denial-of-service attack, the developer can purchase third-party DDoS mitigation software that thoroughly analyzes incoming traffic in an attempt to distinguish bot requests from legitimate user requests.

Buffer Overflow

A buffer overflow attack (also known as a stack smashing or stack overflow attack) is an attack that is very difficult for hackers to execute, as it generally requires a lot of guessing and a lot of luck. The goal of a buffer overflow attack is to hack into the operating system of one of the web application's servers. The attack is executed by injecting binary—also known as bytecode or machine code—into the web application's *run-time stack*. The run-time stack is part of the server's RAM (memory) that is allocated to store temporary data, including flow control data, for the web application. The hacker tries to overwrite part of the run-time stack with executable bytecode, then overwrite the stack's flow control data to make the application execute the injected bytecode. If a hacker executes the attack correctly, the hacker may gain access to all data stored on the application's servers: usernames and passwords, source code, database credentials, etc. Since this attack targets the server's operating system, keeping the operating system up-to-date is an important part of preventing a buffer overflow attack. It is also important to keep code libraries and compilers up-to-date (Frykholm, 2000).

Conclusion

Protecting a web application is not simple, but it can be critical to the application's success. Hackers use a wide variety of attacks to access sensitive data, and it is important for developers to understand how those attacks work and how to effectively defend against them. By taking time during the initial development phase to learn about and implement security features, web application developers will save their time, their money, their customers, and their data in the long run.

References

Banjo, S. (2014). Home Depot hackers exposed 53 million email addresses. *The Wall Street Journal*. Retrieved from http://www.wsj.com/articles/home-depot-hackers-used-password-stolen-from-vendor-1415309282

Code Spaces. (2014). Code Spaces: Is down! *Code Spaces*. Retrieved June 18, 2014, from http://www.codespaces.com/

Duncan, G. (2013). Here's why your email is insecure and likely to stay that way. *Digital Trends*. Retrieved from http://www.digitaltrends.com/computing/can-email-ever-be-secure/#:rHaYKy9xLnMJFA

Frykholm, N. (2000). Countermeasures against buffer overflow attacks. *RSA Laboratories*. Retrieved from http://www.isiloniq.com/emc-plus/rsa-labs/historical/countermeasures-against-buffer-overflow-attacks.htm

Riley, M., Elgin, B., Lawrence, D., & Matlack, C. (2014). Missed alarms and 40 million stolen credit card numbers: How Target blew it. *Bloomberg*. Retrieved from http://www.bloomberg.com/news/articles/2014-03-13/target-missed-warnings-in-epic-hack-of-credit-card-data

Keywords

security, web application, HTTPS, cybersecurity, encryption, email, authentication, authorization, client, server, code injection, SQL, data, internet, packet sniffing, DDoS, buffer overflow, password cracking