Web Security

December 12, 2016

1 Injection Vulnerabilities

Injection vulnerabilities are security flaws in a system in which the system expects some form of user input and an attacker is able to craft inputs which cause code to be run by the system. They are often the result of not properly sanitizing and verifying user inputs, and are possible because in many situations the input given by a user is then embedded directly into the system's code, such as when adding user input to a database keeping track of such inputs, using user input to display something on the webpage, which requires the adding the input to the HTML or JavaScript, or using user input as an argument to a command in a script. In all such cases, an attacker can craft an input which terminates the code in which it's embedded and is followed by malicious code; when the code is compiled with this input, the computer will interpret the attacker's input as part of the code, closing the intended statement and instead running the attacker's input statements.

1.1 SQL Injection

One of the most common code injection vulnerabilities is SQL injection, and in general, the injection of code into databases. Often, databases interact frequently with user input, such as keeping track of certain user input (e.g. usernames), etc. In a typical system, the web server will recieve the user input over the Internet, and use the input to make a SQL query to the database. If the inputs are embedded in SQL queries without being first sanitized or checked for malicious intent, the attacker can inject specifically crafted SQL code which can either read from or modify the database.

(Example) Consider the following straightforward Python code for authenticating users' login information by checking against a SQL database.

```
import sys, sqlite3

conn = sqlite3.connect("users.db")
c = conn.cursor()

user, pass = sys.argv[1 : 3]
valid_user = c.execute("select * from USERS where user = '" + user + "' and \\
                                                  password = '" + pass + "'")
```

The above code directly embeds the user input into a SQL query. If an attacker entered the input

```
' or 1 = 1 --
```

for the user field, when the input was embedded into the SQL query, it would be parsed in such a way as to allow the attacker access. This is because the injected code would be interpreted by the SQL compiler as first closing the string argument to the user field, adding a conjunctive boolean statement which will evaluate to true, and commenting the rest of the query out. Notice that the `1 = 1` part of the input is arbitrary. The attacker could instead begin his input with a single quote followed by a semicolon, so the compiler would interpret the previous query as finished, followed by any arbitrary code to modify or delete the database (followed by the commenting out).

The above example depicts a very common and straightforward pattern of code used when web servers interact with databases using user input, and there have been many instances of SQL injection vulnerabilities being exploited in the real world. Such attacks can be prevented by sanitizing the user input before embedded it in a query. *Sanitization* here refers to the escaping of any special characters, such as qoutes or semicolons, that might be misinterpreted by the SQL compiler, or even the complete disallowing of certain characters or keywords.

In practice, there are many web frameworks which make it possible for web servers to interact with database without handling raw SQL queries, sanitizing all inputs in the background automatically. These solutions, though imperfect, can prevent against a large amount of injection vulnerabilities in general, beyond SQL injection.

2 Cross-site scripting

A ubiquitous web security measure implemented in the early stages of the Internet is known as the *same-origin policy*.

Same-Origin Policy: A standard rule enforced by all web browsers that restricts JavaScript contained in one webpage from only accessing data in other webpages that have the same **origin**. The origin of a webpage is defined as some combination (this is implementation specific) of the URL scheme, the host name, and the port number. Thus, webpages may employ JavaScript but cannot access data from other webpages except those belonging to the same server or site. This prevents websites from accessing data they're not supposed to, and logically separates web pages into their own restricted containers.

Cross-site scripting, or XSS, is a type of injection vulnerability which circumvents the same-origin policy. Unlike SQL injection above, the purpose of XSS is to target other users of the website, rather than directly target the site itself. The attacker injects malicious code into the server, which the server then inadvertently serves to other users, whose browsers then automatically run the code. This allows the attacker to run arbitrary code on victims' browsers, giving them a lot of control over the victim's system. The specifics of XSS can be broadly divided into two categories.

Stored XSS: This is the more dangerous of the two, and involves the attacker managing to non-temporarily store malicious code on the server, such as in a database, that's then handed to every subsequent user who requests a resource containing the code from the server.

A common example setting in which stored XSS is very effective are web environments where users' content is displayed to other users, such as a comment area, a series of blog posts, etc. If an attacker can fool the server into accepting malicious code under the guise of user content, such as a comment, then whenever users request a page with that comment on it, the server will send back a webpage containing the malicious code, thinking it was an ordinary comment, which will then be executed by the victim's browser.

Reflected XSS: In this form of XSS, uses the web server to reflect malicious code to specific target victims. The idea rests on situations in which the web server sends back verbatim user input, HTTP parameters, or other data specified by the client. Such situations are quite common, the canonical example being invalid searches, which often result in an error message of the form `The search "S" was not found`. If S were actually JavaScript code, what would happen is the server would send back a webpage with S embedded in it, thinking it was text, but the browser would parse the JavaScript as code part of the webpage and execute it.

Thus, if an attacker got a victim to click a link which was crafted to have arbitrary JavaScript code reflected back by the server, the victim's browser would make the HTTP request, the server would reply with the webpage containing the JavaScript, and the victim's browser would execute the code. In many cases, the attacker does'nt even need to get the victim to click on the link; rather, simply visiting a webpage is often enough, since there are HTML elements, such as images, which, when parsed by the browser, automatically make HTTP requests. For example, if an attacker put an image tag in a webpage whose source was the reflection URL, the browser would automatically try to load the URL upon receiving the webpage.

As with SQL injection, XSS can be prevented by either browsers or servers escaping JavaScript code, so that it would simply be displayed to the user rather than executed. It's also common to only allow certain inputs for HTTP parameters, or ban `<script>` tags or other JavaScript elemens entirely. An extreme measure is to globally disallow any scripts whatsoever, or only allow specific whitelisted scripts, from running at all.

3 Session Management

HTTP is a stateless protocol, which means its designed with the intent of servers and clients communicating in separate, disparate messages. The inability to store state is a problem for modern web applications, which require persistent state to avoid having to, for example, prompt the user for login for every request. The notion of HTTP cookies was conceived to fix this problem.

Cookies: Cookies are a way of maintaining state across multiple HTTP requests. They're small pieces of data that are stored by the client's browser on the client's computer, and sent to the server along with every HTTP request. Thus, they can be used by the server to maintain stateful information, updating and reading clients' cookies at will. One very important type of cookie used in security is the *authentication cookie*, which is used to validate that a user has logged in and is destroyed when the user logs out. This allows the user to log in to an account once, and proceed with elevated permissions for the remainder of the session, rather than having to login again for every subsequent HTTP request. Structurally, cookies are simply key-value pairs with certain metadata attributes. Cookies have an expiration field which sets the duration of the session, after which the cookie is no longer valid, and is ignored by the server.

Cookie *headers* contain the attributes of the cookie, and include attributes such as whether the cookie should be encrypted (`secure` attribute), the expiration time, the domain (which site to send the cookie to), the path (where on the domain server

to send the cookie), etc. The domain sets the *scope* of the cookie, and is restricted to and domain suffix of the URL, except for top level domains (`.com`, `.net`, etc.).

Cookies are tools used by the client and server to maintain the abstract concept of a *session*. Sessions are just sequences of requests and responses between the client and server. They link the requests and responses with some underlying logic, such as allowing the user to remain logged in after authenticating once. Like cookies, sessions have set expiration times. Sometimes *session tokens* are used instead of cookies; they accomplish the same thing, but unlike cookies, which are all stored on the client's browser and sent with every request, session tokens are merely identifiers that allow the server to distinguish which requests are coming from which clients. Any auxiliary information that needs to be stored about specific clients and their sessions, such as authentication information, is stored by the server rather than in cookies.
Cookies and session tokens lead to another form of web vulnerability, however. Similar to XSS, *cross-site request forgery*, or CSRF, exploits active sessions on a victim's browser to get the victim's browser to execute commands as if they were executed by the user. Because cookies and session tokens are automatically sent along with any HTTP requests the browser makes to the corresponding server, if an attacker can get a victim to click a link or otherwise make an HTTP request to a target server during an active session, the session token will be automatically sent with it.

(Example) For example, if the victim is logged into their bank account in another tab, then the session token is stored by the browser. If, in another tab, the attacker gets the victim's browser to make an HTTP request to the bank's website, such as by embedding the request in an image element, the victim's browser will automatically make the request and attach the correct session token to it, fooling the bank server to executing the desired action specified in the request, thinking it came from the genuine user.

Thus, whereas XSS exploits the trust a user has in a website, CSRF exploits the trust websites have in the user's browser. CSRF attacks are particularly deadly, not only because they can allow for arbitrary commands to be executed on a website, but because they're nearly untraceable, since by all accounts it would appear that the user's browser did in fact make the request, so it's difficult to prove that it wasn't the true user who made the request. This type of attack can be prevented through a few different ways. One common mechanism is for the server to verify that the HTTP request was made from the server's own webpage, and not from a third party.
One way this can be accomplished through the use of a *secret token*, which is given to the browser in every HTTP response and which must be presented in the next HTTP request. Ideally, the browser would only have obtained the secret token if the user had browsed to the server's website and performed the action, and wouldn't have it if an attacker tried to perform the action through an HTTP request originating from a third party web page.
Another way this can be accomplished is if the server checks the HTTP referer, which is a field in the HTTP header that was added precisely for security purposes, and details the previous webpage the browser was on when the request was made, i.e. the webpage which "referred" the request.

4 User Interface Attacks

UI based attacks are performed by websites which modify the interface, often using invisible HTML elements and frames or hyperlinks that say one URL but actually point to another, in order to trick the user into performing unintended actions. One common way this is done is known as *clickjacking*, and involves tricking the user into clicking a malicious link or resource masked by a legitimate one. Thus, the user is under the impression that a legitimate resource was clicked, when in fact the attacker's website may have placed an invisible link beneath the resource, or a frame on top, causing the click to inadvertently perform the wrong action. these malicious websites are often visual copies of legitimate ones, and have URLs that are common and easy-to-miss misspellings or typos of legitimate URLs, in an attempt to better fool the user into thinking the web page is legitimate. A webpage may put a legitimate web site in a frame, and overlay specific buttons that are likely to be pressed with invisible hyperlinks, causing the browser to unintentionally make an HTTP request of the attacker's design. Other forms of UI deception include overlaying parts of a legitimate website in a frame with visible elements that make the true content appear different, such as making a banking website frame seem as though a payment amount is different from the true amount. Alternatively, an attacker might use CSS to alter the appearance of the cursor, or shift the displayed cursor somewhere else and plant a fake cursor where the user expects it to be, to avoid detection. The true, shifted cursor might be visually altered to make it more inconspicuous, and the layout of the page could be structured so that a user attempting to click on one resource would actually click on an intended target, such as a dialogue box authorizing the website for access to the computer's webcam and microphone. This is known as *cursorjacking*.

It's possible for a legitimate website to defend against clickjacking and more general UI based attacks by randomizing its UI, making it more difficult to overlay or modify the appearance of specific elements, since its unpredictable where exactly they'll be when the frame is loaded. Certain actions might also prompt dialogue boxes asking for user confirmation, alerting the user that a certain action is about to be performed. An extreme approach some webpages take is known as *framebusting*,

in which they include code in the webpage which makes it impossible for a third party webpage to put the site in a frame. This can be done by using JavaScript to check if the containing (of the code) webpage is at the top of the DOM, and if not, to move it there.