What is CSRF?

Cross site request forgery (CSRF) is a vulnerability where an attacker performs actions while impersonating another user. For example, transferring funds to an attacker's account, changing a victim's email address, or they could even just redirect a pizza to an attacker's address!

Some form of social engineering, like phishing or spoofing, is usually required for this kind of attack to be successful. The attacker typically needs to trick the user into visiting a malicious website for the attack to take place. This malicious website would then contain a request to the targeted website. If the user is authenticated by the targeted website, the request is executed. This attack works because the user's cookies are automatically included in the modified request to a legitimate application. CSRF vulnerabilities occur when vulnerable web apps simply trust the cookies sent by web browsers without further validation.

About this lesson

In this lesson, we will step into the shoes of a financially motivated attacker and craft a CSRF attack on an unsuspecting user trying to complete a bank transfer. After that, we'll dive into why this attack was possible, covering topics like the Same Origin Policy and SameSite cookies. We will finish by implementing CSRF tokens and SameSite Cookies.

CSRF in action

A web application is vulnerable to CSRF if it relies on session cookies to identify users, and doesn't have any other mechanism for validating requests. Additionally, the request we want to exploit needs to have predictable request parameters so that we can create our own request, like the one in this example.

Luckily for us, we've been tipped off about a vulnerable banking application called "Saturn Bank". This app simply uses session cookies to verify requests. Additionally, the session cookies don't have the SameSite attribute. We'll deep dive into what this attribute is later, in short they control how cookies are submitted in cross site requests.

CSRF under the hood

What just happened?

Behind the scenes, when our victims visited our malicious site “saturnbankgiveaway.com”, a POST request was triggered and sent off to the legitimate Saturn Bank application. The JavaScript sitting in the “script” tags ensures that the form is submitted as soon as the user loads the page, without any user interaction required, or the user even noticing what is happening.

```
<html>
  <body>
    <form action="https://saturnbank.com/transfer" method="POST">
      <input type="hidden" bsb="421314" accountNo="1736123125" amount="100" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

The form above creates the following request to the legitimate Saturn Bank application. The request contains the legitimate user’s session cookie, but contains our bank account number!

POST /transfer HTTP/1.1

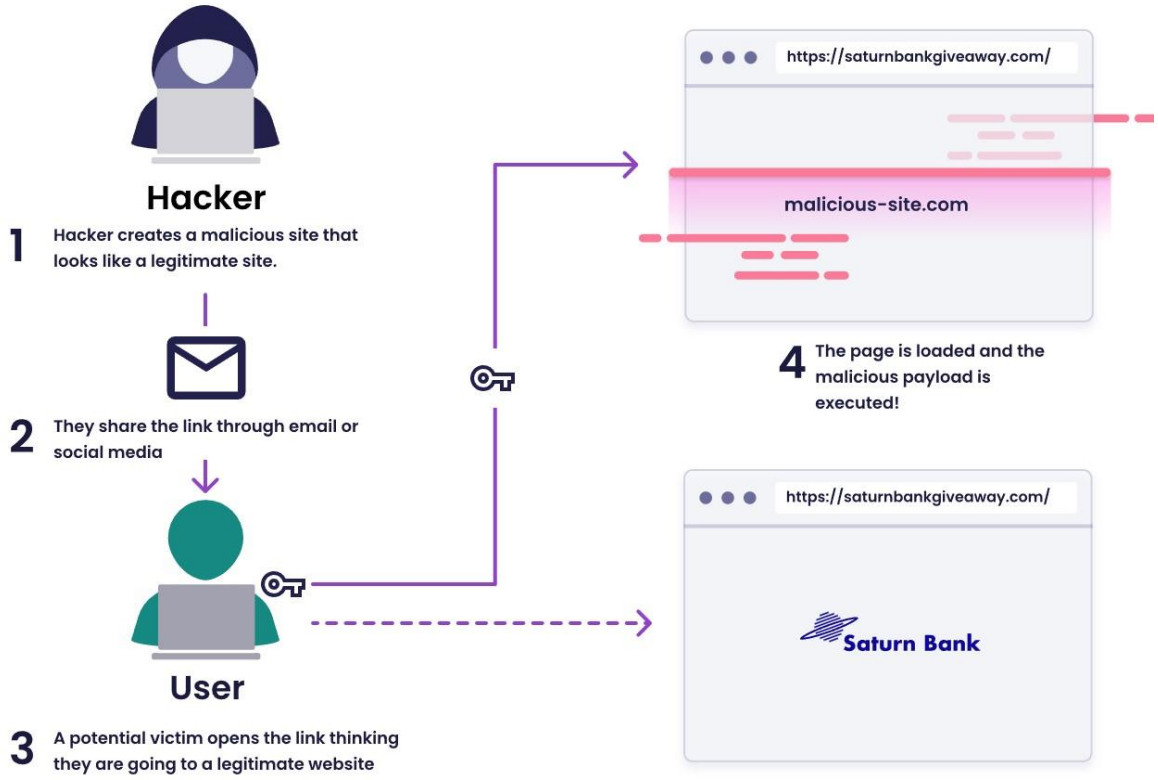
Host: saturnbank.com

Content-Length: 42

Content-Type: application/x-www-form-urlencoded

Cookie: session=OM19vamvikL4yvPQfTqrcjW2ItpDAkDm

bsb=421314&accountNo=1736123125&amount=100



This attack was possible due to a few conditions:

- The user was logged into Saturn Bank
    - The user that visited our site was also logged into the Saturn Bank application. Their session cookie was being stored in their browser, and since it had no SameSite attribute, we were able to steal it for our request
- A state-changing, sensitive 'Action' in the vulnerable app
    - This banking application has a beneficial action for attackers. Being able to control a bank transfer is certainly an attractive proposition!
- Sole reliance on session cookies
    - The application doesn't perform adequate checks to identify a user. It relies solely on the request containing a session cookie
- Standard request parameters
    - There is no unique parameter in the request that the attacker can't determine. This makes it highly repeatable and does not require prior knowledge from the attacker's perspective. For example, if our bank transfer request requires a renewed authentication by asking for the password, attackers cannot perform the CSRF attack

CSRF mitigation

These mitigation techniques can help you defend against CSRF attacks. For simplicity and readability, the following example uses a NodeJS / Express server.

CSRF tokens

One mitigation strategy is to use a random and unique token for use in HTTP requests; these are called CSRF, anti-forgery or request verification tokens. They're a shared secret between the client and server-side of an application, and are included in any requests the client makes to the server. The server validates the token on each request to ensure it's still the authorized user making the request. The token is usually contained in a hidden field of an HTML form. As the token is random and unique, the attacker cannot predict the value for use in their malicious request.

Major web frameworks like Microsoft ASP.NET 5 Razor have CSRF tokens activated by default or include library functions to help. Refer to the OWASP CSRF Cheatsheet for examples on how to implement it in various languages and frameworks. It is advisable to use these or open source libraries. You can research which libraries to use with Snyk Advisor. To show you the underlying concept, we'll implement this example with Double CSRF.

SameSite cookies

Another way you can defend against CSRF is through applying the SameSite attribute to cookies. This attribute is added to the Set-Cookie response header and can be given either the "Strict" or "Lax" values. For example:

Set-Cookie: SessionId=sYMnfCUrAlmqVVZn9dqevxyFpKZt30NN; SameSite=Strict;

The "Strict" value ensures that the browser does not include the cookie in any requests that originate from another site.

The "Lax" value causes the browser to only include the cookie if the request is a GET method and the request was initiated by the user, not scripts.