Web CS 161 Computer Security Notes

Lecture notes by Peyrin Kao

Last update: January 18, 2021

Contact for corrections: Peyrin Kao (peyrin at berkeley.edu)

Disclaimer: These notes are still in beta and haven't been thoroughly fact-checked. In any factual dispute, all other course material takes precedence. Any feedback is welcome.

1 Intro to the Web

It would not be too much of a stretch to say that much of today's world is built upon the Internet. Many of the services that run on top of the Internet come with their own class of vulnerabilities and defenses to match. In particular, we will be focusing on web security, which covers a class of attacks that target web pages and web services.

1.1 URLs

Every resource (webpage, image, PDF, etc.) on the web is identified by a URL (Uniform Resource Locator). A typical URL consists of three parts:

`http://www.example.com/index.html`

The protocol, `http`, tells your browser how to retrieve the resource. In this class, the only two protocols you need to know are HTTP, which we will cover in the next section, and HTTPS, which is a secure version of HTTP using TLS (refer to the networking unit for more details).

The domain name, `www.example.com`, tells your browser which web server to contact to retrieve the resource. Sometimes the domain name will also include a port number, such as `www.example.com:81`, to distinguish between different applications running on the same web server.

The path, `index.html`, tells your browser which page on the web server to request. The web server uses the path to determine which page or resource should be returned to you.

Further reading: What is a URL?

2 HTTP

The protocol that powers the World Wide Web is the Hypertext Transfer Protocol, abbreviated as HTTP. It is the language that clients use to communicate with servers in order to fetch resources and issue other requests. While we will not be able to provide you with a

full overview of HTTP, this section is meant to get you familiar with several aspects of the protocol that are important to understanding web security.

2.1 The Request-Response Model

Fundementally, HTTP follows a request-response model, where clients (such as browsers) must actively start a connection to the server and issue a request, which the server then responds to. This request can be something like "Send me a webpage" or "Change the password for my user account to `foobar`." To the first example, the server would logically respond with the contents of the web page, and to the second example, the response may be something as simple as "Okay, I've changed your password." The exact structure of these requests will be covered in further detail in the next couple sections.

2.2 Structure of a Request

Below is a very simple HTTP request.

```
GET / HTTP/1.1
Host: squigler.com
Dnt: 1
```

The first line of the request contains the method of the request (`GET`), the path of the request (`/`), and the protocol version (`HTTP/1.1`). This is an example of a GET request. Each line after the first line is a request header. In this example, there are two headers, the DNT header and the Host header. There are many HTTP headers defined in the HTTP spec which are used to convey various pieces of information, but we will only be covering a couple of them through this lab.

Here is another HTTP request:

```
POST /login HTTP/1.1
Host: squigler.com
Content-Length: 40
Content-Type: application/x-url-formencoded
Dnt: 1

username=alice@foo.com&password=12345678
```

Here, we have a couple more headers and a different request type: the POST request.

2.3 GET vs. POST

While there are quite a few methods for requests, the two types that we will focus on for this course are GET requests and POST requests. GET requests are intended for "getting" information from the server and generally do not change anything on the server's end. POST requests are intended for sending information to the server that somehow modifies its internal state, such as adding a comment in a forum or changing your password.

Of note, only POST requests can contain a body in addition to request headers. Notice that the body of the second example request contains the username and password that the user `alice` is using to log in. While GET requests cannot have a body, it can still pass query parameters via the URL itself. Such a request might look something like this:

```
GET /posts?search=security&sortby=popularity
Host: squigler.com
Dnt: 1
```

In this case, there are two query parameters, `search` and `sortby`, which have values of `security` and `popularity`, respectively.

3 Elements of a Webpage

The HTTP protocol could only return plain text files, but to make the web more interesting, we write webpages with three different languages that provide additional functionality.

3.1 HTML

HTML (Hypertext Markup Language) lets us create structured documents with paragraphs, links, fillable forms, and embedded images, among other features. You are not expected to know HTML syntax for this course, but some basics are useful for some of the attacks we will cover.

Here are some examples of what HTML can do:

- Create a link to Google: `<a href="http://google.com">Click me</a>`
- Embed a picture in the webpage: `<img src="http://cs161.org/picture.png">`
- Include Javascript in the webpage: `<script>alert(1)</script>`
- Embed the CS161 webpage in the webpage: `<iframe src="http://cs161.org"></iframe>`

Frames pose a security risk, since the outer page is now including an inner page that may be from a different, possibly malicious source. To protect against this, modern browsers enforce frame isolation, which means the outer page cannot change the contents of the inner page, and the inner page cannot change the contents of the outer page.

3.2 CSS

CSS (Cascading Style Sheets) lets us modify the appearance of an HTML page by using different fonts, colors, and spacing, among other features. You are not expected to know CSS syntax for this course.

3.3 Javascript

Javascript is a programming language that runs in your browser. It is a very powerful language–in general, you can assume Javascript can arbitrarily modify any HTML or CSS on a webpage. Webpages can include Javascript in their HTML to allow for dynamic features such as interactive buttons.

Because Javascript is so powerful, modern web browsers typically run Javascript in a sandbox so that any code from a webpage cannot access sensitive data on your computer.

4 Same-Origin Policy

Browsing multiple webpages poses a security risk. For example, if you have a malicious website (`www.evil.com`) and Gmail (`www.gmail.com`) open, you don't want the malicious website to be able to access any sensitive emails or send malicious emails with your identity.

Modern web browsers defend against these attacks by enforcing the same-origin policy, which isolates every webpage in your browser, except for when two webpages have the same origin.

4.1 Origins

The origin of a webpage is determined by its protocol, domain name, and port. For example, the following URL has protocol `http`, domain name `www.example.com`, and port `81`.

```
http://www.example.com/index.html
```

To check if two webpages have the same origin, the same-origin policy performs string matching on the protocol, domain, and port. Two websites have the same origin if their protocols, domains, and ports all match.

Some examples of the same origin policy:

- `http://wikipedia.org/a/` and `http://wikipedia.org/b/` have the same origin. The port (`http`), domain (`wikipedia.org`), and port (none), all match. Note that the paths are not checked in the same-origin policy.
- `http://wikipedia.org` and `http://www.wikipedia.org` do not have the same origin, because the domains (`wikipedia.org` and `www.wikipedia.org`) are different.
- `http://wikipedia.org` and `https://wikipedia.org` do not have the same origin, because the protocols (`http` and `https`) are different.
- `http://wikipedia.org:81` and `http://wikipedia.org:82` do not have the same origin, because the ports (`81` and `82`) are different.

If a port is not specified, the port defaults to 80. This means `http://wikipedia.org` has the same origin as `http://wikipedia.org:80`, but it does not have the same origin as `http://wikipedia.org:81`.

4.2 Exceptions

In general, the origin of a webpage is defined by its URL. However, there are a few exceptions to this rule:

- Javascript runs with the origin of the page that loads it. For example, if you include `<script src="http://google.com/tracking.js></script>` on `http://cs161.org`, the script has the origin of `http://cs161.org`.
- Images have the origin of the page that loads it. For example, if you include `<image src="http://google.com/logo.jpg>` on `http://cs161.org`, the image has the origin of `http://cs161.org`.
- Frames have the origin of the URL where the frame is retrieved from, not the origin of the website that loads it. For example, if you include `<iframe src="http://google.com"></iframe>` on `http://cs161.org`, the frame has the origin of `http://google.com`.

Javascript has a special function, `postMessage`, that allows webpages from different origins to communicate with each other. However, this function only allows very limited functionality.

Further reading: Same-origin policy

5 SQL Injection

5.1 Code Injection

SQL injection is a special case of a more broad category of attacks called code injections.

As an example, consider a calculator website that accepts user input and calls `eval` in the backend to perform the calculation. For example, if a user types `2+3` into the website, the server will run `eval(2+3)` and return the result to the user.

If the web server is not careful about checking user input, an attacker could provide a malicious input like

```
2+3); system("rm *.*"
```

When the web server plugs this into the `eval` function, the result looks like

```
eval(2+3); system("rm *.*")
```

If interpreted as code, this causes the web server to delete all its files!

The general idea behind these attacks is that a web server uses user input as part of the code it runs. If the input is not properly checked, an attacker could create a special input that causes unintended code to run on the server.

5.2 SQL Injection Example

Many modern web servers use SQL databases to store information such as user logins or uploaded files. These servers often allow users to interact with the database through HTTP requests.

For example, consider a website that stores a SQL table of course evaluations named `evals`:

id	course	rating
1	cs61a	4.5
2	cs61b	4.4
3	cs161	5.0

A user can make an HTTP GET request for a course rating through a URL:

```
http://www.berkeley.edu/evals?course=cs61a
```

To process this request, the server performs a SQL query to look up the rating corresponding to the course the user requested:

```
SELECT rating FROM evals WHERE course = "cs61a"
```

Just like the code injection example, if the server does not properly check user input, an attacker could create a special input that allows arbitrary SQL code to be run. Consider the following malicious input:

```
garbage"; SELECT * FROM passwords WHERE username = "admin
```

When the web server plugs this into the SQL query, the resulting query looks like

```
SELECT rating FROM evals WHERE course = "garbage";
SELECT password FROM passwords WHERE username = "admin"
```

If interpreted as code, this causes the query to return the password for the `admin` user!

5.3 SQL Injection Strategies

Writing a malicious input that creates a syntactically valid SQL query can be tricky. Let's break down each part of the malicious input from the previous example:

- `garbage` is a garbage input to the intended query so that it doesn't return anything.
- `"` closes the opening quote from the intended query. Without this closing quote, the rest of our query would be treated as a string, not SQL code.
- `;` ends the intended SQL query and lets us start a new SQL query.
- `SELECT password FROM passwords WHERE username = "admin` is the malicious SQL query we want to execute. Note that we didn't add a closing quote to `"admin`, because the intended SQL query will automatically add a closing quote at the end of our input.

Consider another vulnerable SQL query. This time, we have a `users` table that contains the `username` and `password` of every user.

When the web server receives a login request, it creates a SQL query by plugging in the username and password from the request. For example, if you make a login request with username `alice` and password `password123`, the resulting SQL query would be

```
SELECT username FROM users  WHERE username = "alice"
                            AND password = "password123"
```

If the query returns more than 0 rows, the server registers a successful login.

Suppose we want to login to the server, but we don't have an account, and we don't know anyone's username. How might we achieve this using SQL injection?

First, in the username field, we should add a dummy username and a quote to end the opening quote from the original query:

```
SELECT username FROM users  WHERE username = "alice"
                            " AND password = "____"
```

Next, we need to add some SQL syntax so that this query returns more than 0 rows (since we don't know if `alice` is a valid username). One trick for forcing a SQL query to always return something is to add some logic that always evaluates to true, such as `OR 1=1`:

```
SELECT username FROM users  WHERE username = "alice" OR 1=1
                           " AND password = "____"
```

Next, we have to add some SQL so that the rest of the query doesn't throw a syntax error. One way of doing this is to add a semicolon (ending the previous query) and write a dummy query that matches the remaining SQL:

```
SELECT username FROM users WHERE username = "alice" OR 1=1;
SELECT username FROM users WHERE username = "alice" AND password = "____"
```

The second query might not return anything, but the first query will return a nonzero number of entries, which lets us perform a login. The last step is to add some garbage as the password:

```
SELECT username FROM users WHERE username = "alice" OR 1=1;
SELECT username FROM users WHERE username = "alice" AND password = "garbage"
```

Thus, our malicious username and password should be

```
username = alice" OR 1=1; SELECT username FROM users WHERE username = "alice
password = garbage
```

Another trick to make SQL injection easier is the `--` syntax, which starts a comment in SQL. This tells SQL to ignore the rest of the query as a comment.

In our previous example, we can instead start a comment to ignore parts of the query we don't want to execute:

```
SELECT username FROM users  WHERE username = "alice" OR 1=1
                            --" AND password = "garbage"
```

Thus, another malicious username and password is

```
username = alice" OR 1=1--
password = garbage
```

Further reading: SQL Injection Attacks by Example

5.4 Defense: Escape Inputs

One way of defending against SQL injection is to escape any potential input that could be used in an attack. Escaping a character means that you tell SQL to treat this character as part of the string, not actual SQL syntax.

For example, the quote `"` is used to denote the end of a string in SQL. However, the escaped quote ¨ is treated as a literal quote character in SQL, and it does not cause the current string to end.

By properly replacing characters with their escaped version, malicious inputs such as the ones we've been creating will be treated as strings, and the SQL parser won't try to run them as actual SQL commands.

For example, in the previous exploit, if the server replaces all instances of the quote `"` and the dash `-` with escaped versions, the SQL parser will see

```
SELECT username FROM users  WHERE username = "alice\" OR 1=1\-\-"
                            AND password = "garbage"
```

The escaped quote won't cause the `username` string to end, and the escaped dashes won't cause a comment to be created. The parser will try to look up someone with a username `alice" OR 1=1--` and find nothing.

However, we have to be careful with escaping. If an attacker inputs a backslash followed by a quote `\"`, the escaper might escape the quote and give the input `\\"` to the SQL parser. The parser will treat the two backslashes `\\` as an escaped backslash, and the quote won't be escaped!

The key takeaway here is that building a good escaper can be tricky, and there are many edge cases to consider.

5.5 Defense: Parameterized SQL/Prepared Statements

A better defense against SQL injection is to use parameterized SQL or prepared statements. This type of SQL compiles the query first, and then plugs in user input after the query has already been interpreted by the SQL parser. Because the user input is added after the query is compiled and interpreted, there is no way for any attacker input to be treated as SQL code. Parameterized SQL prevents all SQL injections attacks, so it is the best defense against SQL injection!

In practice, most modern SQL libraries support parameterized SQL and prepared statements.

Further reading: OWASP Cheat Sheet on SQL Injection

6 Cross-Site Scripting (XSS)

XSS is a class of attacks where an attacker injects malicious Javascript onto a webpage. When a victim user loads the webpage, the user's browser will run the malicious Javascript.

XSS attacks are powerful because they subvert the same-origin policy. Normally, an attacker can only run Javascript on websites they control (such as `http://evil.com`), so their Javascript cannot affect websites with origins different from `http://evil.com`. However, if the attacker can inject Javascript into `http://google.com`, then when a user loads `http://google.com`, their browser will run the attacker's Javascript with the origin of `http://google.com`.

There are two main categories of XSS attacks: stored XSS and reflected XSS.

6.1 Stored XSS

In a stored XSS attack, the attacker finds a way to persistently store malicious Javascript on the web server. When the victim loads the webpage, the web server will load this malicious Javascript and display it to the user.

A classic example of stored XSS is a Facebook post. When a user makes a Facebook post, the contents of the post are stored on Facebook's servers, so that other users can load their friends' posts. If Facebook doesn't properly check user inputs, an attacker could make a post that says

```
<script>alert("XSS attack!")</script>
```

This post is now stored in Facebook's servers. If another user loads the attacker's posts, they will receive an HTML page with this script on it, and the browser will run the script and trigger a pop-up that says `XSS attack!`

6.2 Reflected XSS

In a reflected XSS attack, the attacker finds a vulnerable webpage where the server receives user input in an HTTP request and displays the user input in the response.

A classic example of reflected XSS is a Google search. When you make an HTTP GET request for a Google search, such as `https://www.google.com/search?&q=cs161`, the returned webpage with search results will include something like

You searched for: cs161

If Google does not properly check user input, an attacker could create a malicious URL `https://www.google.com/search?&q=<script>alert("XSS attack!")</script>`. When the victim loads this URL, Google will return

You searched for: <script>alert("XSS attack!")</script>

The victim's browser will run the script and trigger a pop-up that says `XSS attack!`

6.3 Defense: Sanitize Input

A good defense against XSS is checking for malicious input that might cause Javascript to run, such as `<script>` tags. However, it is very difficult to write a good detector that catches all XSS attacks. For example, the following input causes Javascript to run without ever using `<script>` tags:

```
<img src=1 href=1 onerror="javascript:alert("XSS attack!")"></img>
```

Just like SQL input escaping, sanitizing potentially dangerous input can be very tricky. For example, consider an escaper that searches for all instances of `<script>` and `</script>` and removes them. An attacker could provide this malicious input:

```
<scr<script>ipt>alert("XSS attack!")</scr<script>ipt>
```

After the escaper removes the two `<script>` tags it sees, the result is `<script>alert("XSS attack!")</script>`, and the attacker can still execute Javascript!

Another way to escape input is to replace potentially dangerous characters with their HTML encoding. For example, the less than (<) and greater than (>) signs are encoded as `&lt;` and `&gt;`, respectively. These encodings cause less than and greater than signs to display on the webpage, without being interpreted as HTML.

6.4 Defense: Content Security Policy

Another XSS defense is using a content security policy (CSP) that specifies a list of allowed domains where scripts can be loaded from. For example, `cs161.org` might allow scripts that are loaded from `*.cs161.org` or `*.google.com` and disallow all other scripts, including any inline scripts that are injected by the attacker.

CSPs are defined by a web server and enforced by a browser. In the HTTP response, the server attaches a `Content-Security-Policy` header, and the browser checks any scripts against the header.

: OWASP Cheat Sheet on XSS

7 Cookies and Session Management

HTTP is a stateless protocol, which means each request and response is independent from all other requests and responses. However, many features on the web require maintaining some form of state. For example, when you log into your email account, you can stay logged in across many requests and responses. If you enable dark mode on a website and make subsequent requests to the website, you want the pages returned to have a dark background. If you're browsing an online shopping website, you want the items in your cart to be saved

across many requests and responses. Browser and servers store HTTP cookies to support these features.

At a high level, you can think of cookies as pieces of data stored in your browser. When you make a request to enable dark mode or add an item to your shopping cart, the server sends a response with a `Set-Cookie` header, which tells your browser to store a new cookie. These cookies encode state that should persist across multiple requests and responses, such as your dark mode preference or a list of items in your shopping cart. In future requests, your browser will automatically attach the relevant cookies to a request and send it to the web server. The additional information in these cookies helps the web server customize its response.

7.1 Cookie Attributes

Every cookie is a name-value pair. For example, a cookie `darkmode=true` has name `darkmode` and value `true`.

For security and functionality reasons, we don't want the browser to send every cookie in every request. A user might want to enable dark mode on one website but not on another website, so we need a way to only send certain cookies to certain URLs. Also, as we'll see later, cookies may contain sensitive login information, so sending all cookies in all requests poses a security risk. These additional cookie attributes help the browser determine which cookies should be attached to each request.

- The `Domain` and `Path` attributes tell the browser which URLs to send the cookie to. See the next section for more details.
- The `Secure` attribute tells the browser to only send the cookie over a secure HTTPS connection.
- The `HttpOnly` attribute prevents Javascript from accessing and modifying the cookie.
- The `expires` field tells the browser when to stop remembering the cookie.

7.2 Cookie Policy: Domain and Path

The browser sends a cookie to a given URL if the cookie's `Domain` attribute is a domain-suffix of the URL domain, and the cookie's `Path` attribute is a prefix of the URL path. In other words, the URL domain should end in the cookie's `Domain` attribute, and the URL path should begin with the cookie's `Path` attribute.

For example, a cookie with `Domain=example.com` and `Path=/some/path` will be included on a request to `http://foo.example.com/some/path/index.html`, because the URL domain ends in the cookie domain, and the URL path begins with the cookie path.

7.3 Cookie Policy: Setting Domain and Path

For security reasons, we don't want a malicious website `evil.com` to be able to set a cookie with domain `bank.com`, since this would allow an attacker to affect the functionality of the legitimate bank website. To prevent this, the cookie policy specifies that when a server sets a cookie, the cookie's domain must be a URL suffix of the server's URL. In other words, for the cookie to be set, the server's URL must end in the cookie's `Domain` attribute. Otherwise, the browser will reject the cookie.

For example, a webpage with domain `eecs.berkeley.edu` can set a cookie with domain `eecs.berkeley.edu` or `berkeley.edu`, since the webpage domain ends in both of these domains.

This policy has one exception: cookies cannot have domains set to a top-level domain, such as `.edu` or `.com`, since these are too broad and pose a security risk. If `evil.com` could set cookies with domain `.com`, the attacker would have the ability to affect all `.com` websites, since this cookie would be sent to all `.com` websites.

The cookie policy allows a server to set the `Path` attribute without any restrictions.

Further reading: Cookies

7.4 Session Management

Cookies are often used to keep users logged in to a website over many requests and responses. When a user sends a login request with a valid username and password, the server will generate a new session token and send it to the user as a cookie. In future requests, the browser will attach the session token cookie and send it to the server. The server maintains a mapping of session tokens to users, so when it receives a request with a session token cookie, it can look up the corresponding user and customize its response accordingly.

Secure session tokens should be random and unpredictable, so an attacker cannot guess someone else's session token and gain access to their account. Many servers also set the `HttpOnly` and `Secure` flags on session tokens to protect them from being accessed by XSS vulnerabilities or network attackers, respectively.

It is easy to confuse session tokens and cookies. Session tokens are the values that the browser sends to the server to associate the request with a logged-in user. Cookies are how the browser stores and sends session tokens to the server. Cookies can also be used to save other state, as discussed earlier. In other words, session tokens are a special type of cookie that keep users logged in over many requests and responses.

8 Cross-Site Request Forgery (CSRF)

Using cookies and session tokens to keep a user logged in has some associated security risks. In a cross-site request forgery (CSRF) attack, the attacker forces the victim to make an

unintended request. The victim's browser will automatically attach the session token cookie to the unintended request, and the server will accept the request as coming from the victim.

For example, suppose a website has an endpoint `http://example.com/logout`. To log out, a user makes a GET request to this URL with the appropriate session token attached, and the server checks the session token and performs the logout. If an attacker can trick a victim into clicking this link, the victim will be logged out of the website without their knowledge.

CSRF attacks can also be executed on URLs with more malicious actions. For example, a GET request to `https://bank.com/transfer?amount=100&recipient=mallory` with a valid session token might send $100 to Mallory. An attacker could send an email to the victim with the following HTML snippet:

```
<img src="https://bank.com/transfer?amount=100&recipient=mallory>
```

This will cause the browser to try and fetch an image from the malicious URL by making a GET request. Because the browser automatically attaches the session token to the request, this causes the victim to unknowingly send $100 to Mallory.

It is usually bad practice to have HTTP GET endpoints that can change server state, so this type of CSRF attack is less common in practice. However, CSRF attacks are still possible over HTTP POST requests. HTML forms are a common example of a web feature that generates HTTP POST requests. The user fills in the form fields, and when they click the Submit button, the browser generates a POST request with the filled-out form fields. Consider the following HTML snippet on an attacker's webpage:

```
<form name=evilform action=https://bank.com/transfer>
  <input name=amount value=100>
  <input name=recipient value=mallory>
</form>
<script>document.evilform.submit();</script>
```

When the victim visits the attacker's website, this HTML snippet will cause the victim's browser to make a POST request to `https://bank.com/transfer` with form input values that transfer $100 to Mallory. Like before, the victim's browser automatically attaches the session token to the request, so the server accepts this POST request as if it was from the victim.

8.1 Defense: CSRF Token

A good defense against CSRF attacks is to include a CSRF token on webpages. When a legitimate user loads a webpage from the server with a form, the server will randomly generate a CSRF token and include it as an extra field in the form. (In practice, this field often has a hidden attribute set so that it's only visible in the HTML, so users don't see random strings every time they submit a form.) When the user submits the form, the form will include the CSRF token, and the server will check that the CSRF token is valid. If the CSRF token is invalid or missing, the server will reject the request.

To implement CSRF tokens, the server needs to generate a new CSRF token every time

a user requests a form. CSRF tokens should be random and unpredictable so an attacker cannot guess the CSRF token. The server also needs to maintain a mapping of CSRF tokens to session tokens, so it can validate that a request with a session token has the correct corresponding CSRF token. This may require the server to store a large amount of state if it expects heavy traffic.

If an attacker tries the attack in the previous section, the malicious form they create on their website will no longer contain a valid CSRF token. The attacker could try querying the server for a CSRF token, but it would not properly map to the victim's session token, because the victim never requested the form legitimately.

8.2 Defense: Referer Validation

Another way to defend against CSRF tokens is to check the Referer field in the HTTP header. When a browser issues an HTTP request, it includes a Referer header which indicates which URL the request was made from. For example, if a user fills out a form from a legitimate bank website, the Referer header will be set to `bank.com`, but if the user visits the attacker's website and the attacker fills out a form and submits it, the Referer header will be set to `evil.com`. The server can check the Referer header on each request and reject any requests that have untrusted or suspicious Referer headers.

Referer validation is a good defense if it is included on every request, but it poses some problems if someone submits a request with the Referer header left blank. If a server accepts requests with blank Referer headers, it may be vulnerable to CSRF attacks, but if a server rejects requests with blank Referer headers, it may reduce functionality for some users.

In practice, Referer headers may be removed by the browser, the operating system, or a network monitoring system for privacy issues. For example, if you click on a link to visit a website from a Google search, the website can know what Google search you made to visit its website from the Referer header. Some modern browsers also have options that let users disable sending the Referer header on all requests. Because not all requests are guaranteed to have a Referer header, it is usually only used as a defense-in-depth strategy in addition to CSRF tokens, instead of as the only defense against CSRF attacks.

Further reading: OWASP Cheat Sheet on CSRF