CS155: Computer and Network Security **Spring 2023**

Homework #2

Due: Thursday, May 25, 2023, by Gradescope.
Julian Cooper

Problem 1. Same Origin Policy We discussed in lecture how the DOM same-origin policy defines an origin as the triple (protocol, domain, port). Explain what would go wrong if the DOM same-origin policy were only defined by domain, and nothing else. Give a concrete example of an attack that a network attacker can do in this case, but cannot do when using the standard definition of the same-origin policy.

Idea: If the DOM same-origin policy were only defined by domain, then a network attacker could create a malicious website using the same domain but a differnt protocol (http instead of https) or port. In particular,

- Attacker creates a malicious website `http://bank.com` that looks like the real bank website `https://bank.com`.
- Imagine that a user is logged into their bank account at `https://bank.com` and visits a malicious version of the website `http://bank.com`.
- The macilious version of the website could then steal the user's session cookie and use it send post requests to user's real bank account.

.

Problem 2. Cross Site Script Inclusion (XSSI) Attacks

Consider a banking web site bank.com where after login the user is taken to a user information page
`https://bank.com/accountInfo.html`

The page shows the user's account balances. Here `accountInfo.html` is a static page: it contains the page layout, but no user data. Towards the bottom of the page a script is included as:
`<script src="//bank.com/userdata.js"></script> (*)`

The contents of `userdata.js` is as follows:

```
displayData({"name":  "John Doe", "AccountNumber":  12345, "Balance":  45})
```

The function `displayData` is defined in `accountInfo.html` and uses the provided data to populate the page with user data.

The script `userdata.js` is generated dynamically and is the only part of the page that contains user data. Everything else is static content.

Suppose that after the user logs in to his or her account at `bank.com` the site stores the user's session token in a browser cookie.

a. Consider user `John Doe` who logs into his account at `bank.com` and then visits the URL `https://evil.com/`. Explain how the page at `evil.com` can cause all of John Doe's data to be sent to `evil.com`. Please provide the code contained in the page at `evil.com`.

Idea: Since the function `displayData` is defined in `accountInfo.html`, it is accessible to `evil.com`. Thus, `evil.com` can define its own function `displayData` that sends the data to `evil.com` and then calls the original `displayData` function.

The code for `evil.com` is as follows:

```
<script>
    function displayData(data) {
        console.log("name: ", data.name);
        console.log("account: ", data.AccountNumber);
        console.log("balance: ", data.Balance);
    }
</script>
...
...
<script src="//bank.com/userdata.js"></script>
```

Note, we only need to `console.log` rather than send a post request to `evil.com` since John Doe has already visited `evil.com` and so the browser will automatically

send the console outputs `evil.com`.

b. How would you keep `accountInfo.html` as a static page, but prevent the attack from part (a)? You need only change line (*) and `userdata.js`. Make sure to explain why your defense prevents the attack.
Hint: Try loading the user's data in a way that gives `bank.com` access to the data, but does not give `evil.com` access. In particular, `userdata.js` need not be a Javascript file.

Idea: There are two issues with the original setup: (a) Javascript sends data in plaintext (leaks information for ease of use) and (b) `displayData` can be redefined by the attacker. We can address both issues by loading data as a JSON file directly in `accountInfo.html` rather than via Javascript file.

Code example to replace (*):

```
fetch('//bank.com/userdata.json') // fetch returns a promise
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

By the same-origin policy, `evil.com` will not be able to read contents of our JSON.

Problem 3. HTML Canvas Element

The canvas HTML element creates a 2D rectangular area and lets Javascript draw whatever it wants in that area. Canvas is used for client-side graphics such as drawing a path on a map loaded from Google maps. For the purpose of the associated same-origin policy, the origin of a canvas is the origin of the content that created it. In the map example, the origin of the Javascript that creates the canvas is Google. Canvas lets Javascript read pixels from any canvas in its origin using the `GetImageData()` method.

a. Canvas lets Javascript embed images from any domain in the canvas. Suppose a user has authenticated to a site that displays private information. Describe an attack that would be possible if Javascript from one domain could embed an image from another domain in the canvas and then use `GetImageData()` to read pixels from that image.

Same-origin policy is not checked. The attacker can embed an image from the private site in its canvas then use `GetImageData()` to read the pixels from the image. For example, if the attacker embedded an image of the user's account balance, then the attacker could read the pixels from the image and extract the account balance once user is logged in.

b. How would you restrict `GetImageData()` to prevent the attack above?

Enforce the same-origin policy for `GetImageData()`. This would mean if you load any content to canvas element from another origin, you would not be able to read that part of the canvas.

c. A canvas element can be placed anywhere in the browser content area and can be made transparent so that the underlying content under the canvas shows through. What security problem arises if calling `GetImageData()` always returned the actual pixels shown on the screen at that position? Briefly explain whether your restriction from part (b) prevents this problem and why/why not.

The attacker can place the canvas element over the private content and then use `GetImageData()` to read the pixels from the canvas. No, the defense described in part (b) does not prevent this attack since the pixels of the canvas itself are from the same origin (even though transparent).

d. How would you design `GetImageData()` to defend against the vulnerability from part (c)? Propose a design that does not require the browser to test if the requested pixel is over content from another origin.

We could design the `GetImageData()` function to return all zeros for any parts of the canvas which are transparent, rather than the pixels below. The browser specifies this behavior and the attacker cannot change it.

Problem 4. CSRF Defenses

a. In class we discussed Cross Site Request Forgery (CSRF) attacks against web sites that rely solely on cookies for session management. Briefly explain a CSRF attack on such a site.

CSRF attacks are a type of web exploit where a website transmits unauthorized commands as a user that the web app trusts. For example,

- Attacker designs a self-contained HTML page (e.g. b.html from project 2) will logic to send a transfer post request to the bank.com server.
- Our user logs into bank.com and navigates to the transfer page. Their browser session now contains cookies required to submit a transfer post request.
- Somehow (e.g. phishing email link) the user is tricked into visiting the attacker's HTML page which contains javascript to submit a transfer post request to bank.com.
- This request is made with proper credentials since the user is already logged in within the same browser session!

.

b. A common CSRF defense places a token in the DOM of every page (e.g., as a hidden form element) in addition to the cookie. An HTTP request is accepted by the server only if it contains both a valid HTTP cookie header and a valid token in the POST parameters. Why does this prevent the attack from part (a)?

While the attacker can still send a transfer post request to bank.com, the request will be rejected since the attacker's HTML page does not contain the CSRF token required by the server. This token is session-specific and cannot be accessed by attacker because Same-Origin Policy (SOP) prevents reading DOM content.

c. One approach to choosing a CSRF token is to choose one at random. Suppose a web server chooses the token as a fresh random string for every HTTP response. The server checks that this random string is present in the next HTTP request for that session. Does this prevent CSRF attacks? If so, explain why. If not, describe an attack.

Yes! A random CSRF token regenerated for each HTTP response is a good defense because the attacker cannot predict the token value, nor can they use side-channel attacks to learn the token over multiple requests.

d. Another approach is to choose the token as a fixed random string chosen by the server. That is, the same random string is used as the CSRF token in all HTTP responses from the server over a given time period. Does this prevent CSRF attacks? If so, explain why. If not, describe an attack.

No! A fixed CSRF token is vulnerable to a side-channel attack (timing, etc.). An attacker could make repeated requests to the server and try to learn the session token. He/she would then be able to submit valid (but malicious) post requests for the remaining life of the session (or time period in this case).

e. Why is the Same-Origin Policy important for the cookie-plus-token defense?

The Same-Origin Policy (SOP) prevents the attacker from reading the CSRF token from the DOM. This is important because even if the attacker can use a logged in user's cookie, they cannot read the CSRF token from the DOM and therefore cannot submit a valid post request.

Problem 5. Content Security Policies Recall that content security policy (CSP) is an HTTP header sent by a web site to the browser that tells the browser what it should and should not do as it is processing the content. The purpose of this question is to explore a number of CSP directives. Please use the CSP specification to look up the definition of the directives in the questions below.

a. Explain what the following CSP header does:

`Content-Security-Policy: script-src 'self'`

What is the purpose of this CSP directive? What attack is it intended to prevent?

> Purpose: The script source 'self' directive informs the browser to only execute scripts which are hosted on the same origin as the website.
>
> Attack: This will prevent Cross-Site Scripting (XXS) attacks which rely on injecting Javascript into the victim site since these scripts are from a different origin and so will no longer be permitted to run in the user's browser.

b. What does the following CSP header do: `Content-Security-Policy: frame-ancestors 'none'` What attack does it prevent?

> Purpose: Frame ancestors directives specify the valid parents sites (origins) which can embed a certain webpage. Setting this directive to 'none' tells the browser that the page cannot be displayed in a <iframe>, <frame>, <embed> or <object>. In other words, it cannot be embeeded within another site.
>
> Attack: Helpful for preventing clickjacking attacks where user is tricked into interacting with a fake (malicious) webpage. The attacker embeds content from a legitimate webpage inside the malicious webpage and tracks the user's interactions - e.g. type password into embedded version of bank.com that attacker can read. This does not work if the embedded legitimate site cannot be displayed.

c. What does the following CSP header do:

`Content-Security-Policy: sandbox` 'allow-scripts'

Suppose a page loaded from the domain `www.xyz.com` has the sandbox CSP header, as above. This causes the page to be treated as being from a special origin that always fails the same-origin policy, among other restrictions. How does this impact the page's ability to read cookies belonging to `www.xyz.com` using Javascript? Give an example where a web site might want to use this CSP header.

Purpose: The sandbox directive applies restrictions to a page's actions including preventing pop-ups, execution of plugins and scripts, and enforcing the same-origin policy. Setting this directive to 'allow-scripts' specifically allows scripts, but maintains all other restrictions (e.g. pop-ups, same-origin policy).

Impact: The page will not be able to read cookies from `www.xyz.com` since this would violate the same-origin policy. This is because the sandbox CSP header guarantees that `www.xyz.com` will be treated as a special origin that always fails the same-origin policy.

Example: Whenever a website wants to embed a third party widget that requires scripts to run, but does not want to allow the third party to read cookies from the website. We could image a blog platform where we might want to allows users that ability to include a Twitter feed widget, which would require a script to run. However, we would not want the Twitter widget to be able to read cookies from the blog platform. In this case, we could use the sandbox directive with 'allow-scripts' to allow the widget to run, but in a sandboxed environment where it cannot do as much damage.

Problem 6. Broken password checker Consider the following broken password checking code that runs inside of an SGX enclave. Assume that `*correctPwd` is only visible inside the enclave, and `*enteredPwd` is supplied as an argument from outside the enclave.

```
void check_passwd(char *enteredPwd, *correctPwd) {
    for(i=0, i < LEN, ++i) {
        if (enteredPwd[i] == correctPwd[i]) {
            sleep(.1);   /* sleep for 0.1 sec */
        } else {
            return(-1);  /* disallow login */
        }
    }
    return(0); /* allow login */
}
```

The enclave returns the result of this function to the caller outside the enclave. Assume that the caller ensures that `*givenPwd` and `*correctPwd` are exactly LEN characters.

a. Describe a simple attack that lets a local attacker outside the enclave extract `*correctPwd` from the enclave using at most `LEN * 256` login calls into the enclave.

> Idea: Timing attack! For each character in the password, we can guess and then measure the time it takes for the enclave to return. If the time is longer than the time it takes to return when the character is incorrect, then we know we have guessed the correct character. We can then move on to the next character and repeat the process until we have guessed the entire password. Since each character can take on 256 possible values, we will need to make at most LEN * 256 calls to the enclave.

b. Write a password checker with the same interface as in part (a) that avoids the problem you identified in part (a).

> Idea: Want to make sure that the time it takes to return is independent of the character being checked. We could achieve this by either making sure that the time it takes to return is the same for all characters, or by making sure that the time it takes to return is random. In the code below, I implement the first approach, where the time will be the same regardless of character guessed. Note, it is important to keep some timing delay to reduce feasibility of brute force attacks.
>
> ```
> void check_passwd(char *enteredPwd, *correctPwd) {
>     for(i=0, i < LEN, ++i) {
>         if (enteredPwd[i] == correctPwd[i]) {
>             sleep(.1);   /* sleep for 0.1 sec */
>         } else {
> ```

```
                sleep(.1);   /* sleep for 0.1 sec */
                return(-1);  /* disallow login */
            }
        }
        return(0); /* allow login */
    }
```

Problem 7. Certificate Authorities Every certificate contains a field called CA Flag that is set to 'true' if the public key being certified belongs to a CA and is 'false' otherwise. When the browser verifies a certificate chain for a domain, it checks that all the certificates in the chain have the CA flag set to 'true', except for the leaf certificate for which the CA flag is ignored.

Describe an attack that would be possible if the browser did not do this check; that is, it did not check that the CA flag is set to 'true' for the certificates in the chain.

Idea: An attacker that controls the leaf certificate could act as a Certificate Authority itself by self-signing a certificate and using that certificate to sign other certificates (e.g. for malicious versions of websites).

- The server at the end of the certificate chain is typically not granted Certificate Authority powers since the CA flag is set to 'false' for the leaf certificate. This means it cannot sign other certificates.
- However, if the browser did not check that the CA flag is set to 'true' for the certificates in the chain, then the server at the end of the certificate chain could act as a Certificate Authority itself by self-signing a certificate.
- The newly minted malicious Certificate Authority could then use this certificate to sign other certificates for malicious versions of websites (e.g. a fake gmail).
- Our user might accidentally navigate to one of these malicious websites via typo (gmmail.com) or phishing and the browser would not intervene.

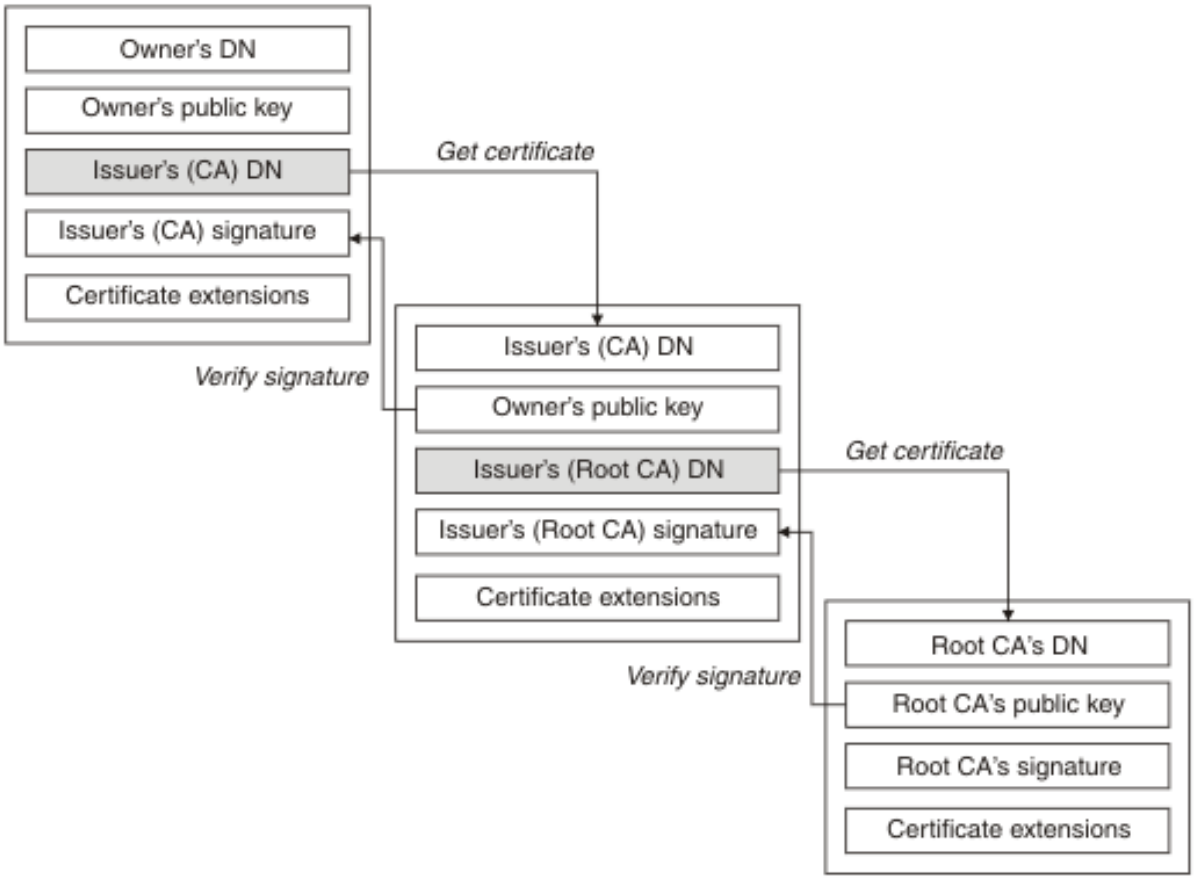

Figure 1: Certificate Authority Chain (Source: www.ibm.com/docs)