CS155: Computer and Network Security

CS155: Homework #2

Spring 2022

Due: Thursday, May 25

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.

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`.

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.

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.

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

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.

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.

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.

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)?

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.

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.

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

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?

b. What does the following CSP header do:

```
Content-Security-Policy: frame-ancestors 'none'
```

What attack does it prevent?

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.

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.

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

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.