Stanford University
Computer Science Department
CS 253 Final Exam Fall 2021

December 7, 2021

This is a closed book exam. You may use 3 (double-sided) sheets of notes. You have 3 hours. Write all of your answers directly on the paper. Make your answers as concise as possible.

NOTE: We will take off points if a correct answer also includes incorrect or irrelevant information (i.e. don't write everything you know in hopes of saying the correct buzzword.)

Question	Score
1. True or False (25 points)	
2. Short Answers (42 points)	
3. The Great Cannon (24 points)	
4. Cookies (9 points)	
5. Coffee Shop Wi-Fi (12 points)	
6. Delete Account (6 points)	
7. User Agent (6 points)	
8. CSP (9 points)	
9. XSSI (12 points)	
TOTAL (145 possible points)	

Stanford University Honor Code

In accordance with both the letter and the spirit of the Honor Code, I did not cheat on this exam nor will I assist someone else in cheating.

Name and SUNet ID:

Signature:

Problem 1. True or False (1 point each)

For each question, write either "True" or "False".

T 1. Reflected XSS occurs when an attacker gets a victim to send a request with malicious input to a server which includes the unsanitized input in the HTML output it produces.

F 2. The CSP directive `strict-dynamic` tells the browser to use HTTPS for all in-site resource requests, regardless of the protocol in the URL.

F 3. Checking the `Referer` header is a robust defense against CSRF attacks.

F 4. Cookies set with the `HttpOnly` attribute are never sent over HTTPS.

T 5. Setting the `SameSite=Lax` attribute on a cookie is a good defense against CSRF.

T 6. The logout process in a web app should mark the session as expired on the server.

F 7. The Same Origin Policy used for the DOM is the same as the Same Origin Policy applied to cookies.

F 8. The server can trust cookie values in HTTP requests to be untampered since the cookies are set by the server.

T 9. A CSRF vulnerability at a bank has the following consequence: a malicious web site can issue requests to the bank on behalf of users visiting the malicious web site.

T 10. The design of HTTPS prevents a network attacker from replaying an old session.

T 11. The cookie attribute `HttpOnly` helps to mitigate the effects of XSS attacks by preventing client-side JavaScript from reading the cookie.

T 12. A malicious website can execute a successful clickjacking attack even if the victim website uses HTTPS and the user's browser correctly implements the same origin policy.

T 13. If `site-a.com` loads a website from another domain, `site-b.com`, inside of an iframe, the same origin policy prevents Javascript from `site-a.com` from accessing any of `site-b.com`'s website content in the iframe.

T 14. `postMessage` is a powerful browser-based communication mechanism that allows any frame to broadcast messages that will be accepted by any other frame, regardless of frame origin.

F 15. The best way to prevent untrusted user input from exploiting your application is to use encryption.

T 16. When the server receives an HTTP request with an `Origin` header, the server can rely on the browser to ensure that the header value reflects the true origin of the page that initiated the HTTP request.

F 17. When a site sends a ***cross-origin*** GET request ***without*** custom headers, the browser will first send a "preflight" OPTIONS request.

F 18. When a site sends a ***same-origin*** PUT request ***with*** custom headers, the browser will first send a "preflight" OPTIONS request.

T 19. In a "recursive DNS query", a client (DNS resolver) queries a single DNS server, which may in turn query other DNS servers on behalf of the requester until the answer can be determined, and then returns the answer to the client.

T 20. HTTPS will protect the user even if an attacker manages to change or spoof a DNS record so that it points to an attacker-controlled IP address.

F 21. When accepting untrusted input from the user, we should escape it *before* it is added to the database so that we can later use it without worrying about escaping.

T 22. Your browser will save cookies even from sites you have not visited directly. ("Visited directly" means that e.g. you navigated to the site and its URL appeared in the browser's address bar)

T 23. One benefit of DNS-over-HTTPS, as compared with normal DNS, is that it provides privacy from network eavesdroppers such as ISPs.

T 24. A local HTTP server can defend against DNS rebinding by inspecting the `Host` header and dropping any request for which the value is not `localhost:<some-port>`.

F 25. A website can use Certificate Transparency to improve the security of their users' HTTPS connections, at the HTTPS protocol level.

Problem 2. Short Answers (2 points each)

For each question, write a short answer using no more than **50 words**.

1. Name the three parts of a URL that are used to determine the URL's origin.

1. Protocol/scheme, hostname, port

2. What is the purpose of the HTTP `Strict-Transport-Security` header (HSTS)? Please make sure to explain what attack is being prevented by this header.

2. The HSTS header forces the browser to use HTTPS instead of HTTP on all future requests. This prevents a passive network attacker from observing the traffic, or an active network attacker from performing an SSLstrip man-in-the-middle attack.

3. (Continued from previous question) What denial-of-service attack could a network attacker perform if the HSTS header was allowed to be set on HTTP responses? (Recall that the browser only respects the HSTS header if it is set on an HTTPS response.)

3. Denial of service. A network attacker could modify an HTTP response and add an HSTS header with a long expiration time. Doing this on a website which does not actually support HTTPS would break the website for this user.

4. (Continued from previous question) The protection offered by the HSTS header only applies after a user has visited the site at least once. What is the mechanism that a site can use to ensure that even the first request that a user makes to the server uses HTTPS?

4. A website can request to be added to the HSTS preload list which is built into all the major browsers.
Alternatively, sites can use the proposed DNS “HTTPS” record as an alternative to HSTS preload.

5. Suppose A and B are two frames in a browser that are loaded from different origins. Why is it a reasonable security policy to allow A to navigate B to another origin based only on whether the display area of A contains the display area of B and A has control over that area?

5. Frame A could remove Frame B from its DOM and add a new frame which loads any arbitrary URL. This is functionally equivalent to simply navigating Frame B. **[NOTE: This question was too confusing, so we gave everyone full credit.]**

6. You are a penetration tester evaluating a website for security vulnerabilities. You notice that their authentication system chooses session IDs by taking the cryptographic hash of the username, using e.g. the SHA256 hash function. For example, a user "feross" would be assigned a session ID of `SHA256('feross')` and a user "kaminsky" would be assigned a session ID of `SHA256('kaminsky')`, and so on. Describe an attack against this authentication system.

6. The attacker can manually set their session ID cookie to the hash of the username they want to login as. Since SHA-256 is a public algorithm, this is trivial for them to do.

7. (Continued from previous question) The website "fixes" the issue by updating the server code so a random counter value between 1 and 2^{256} is chosen at startup and included along with the username before hashing it. The server increments the counter each time it generates a session ID. For example, assume the server randomly chose 9000 as the initial counter value. If "feross" is the first user to login, he will be assigned a session ID of SHA256('feross' + 9000). If "ada" logs in next, she will be assigned a session ID of SHA256('ada' + 9001), and so on. Is this authentication system secure? Write "Secure" or "Not Secure". If you choose "Not Secure", then describe an attack against this authentication system.

7. This design is secure. The counter value is not reversible, since 2^{256} is too large of a state space (as large as the output space of SHA-256 itself!) to search.

   Also acceptable: Not secure. No delimiter between username and counter means that years “feross” and “feross1” can be confused because SHA256(“feross1” + 1) = SHA256(“feross” + 11) will produce the same hash and, therefore, the same session ID.

8. Consider a Web site `xyz.com` that implements a phone dialer. When the user enters a phone number to call, the browser opens a new window to `xyz.com/call.html` containing the following Javascript that defines a `postMessage` event listener:

```
function receiveMessage (event) {
      // event.data is a phone number from sender
      initiatePhoneCallTo(event.data)
}
window.addEventListener('message', receiveMessage)
```

The parent page then sends a `postMessage` to this window to initiate the call. This activates the `receiveMessage` function which makes the call. Explain how an attacker website can cause a visitor to initiate phone calls to arbitrary phone numbers. Assume the visitor is logged in to her `xyz.com` account, but does not have `xyz.com` open in a window.

8. Open a new window to xyz.com/call.html with window.open() and use the window reference to send a postMessage with an arbitrary phone number.

9. (Continued from previous question) If the function `receiveMessage` started with the following line:

```
if (event.origin !== 'https://xyz.com') return
```

Would this eliminate the problem you identified in the previous question? Recall that `event.origin` is the true origin that initiated the `postMessage` call.

9. Yes.

10. An attacker injects an XSS payload into the HTML page sent by your server. Given the following CSP, would the XSS attack succeed? Justify your answer.

    CSP: `Content-Security-Policy: default-src 'self';`

    XSS: `<script>alert(document.cookie)</script>`

    10. No, the attack fails. The default-src 'self' directive means that script content is only allowed from external scripts loaded from the same origin. Inline scripts are blocked unless 'unsafe-inline' is present. Since the XSS attack is an inline script, it is blocked.

11. An attacker injects an XSS payload into the HTML page sent by your server. Given the following CSP, would the XSS attack succeed? Justify your answer.

    CSP: `Content-Security-Policy: script-src 'self' 'nonce-PAk3kslfKFAoaP423';`

    XSS: `<script>alert(document.cookie)</script>`

    11. No, the attack fails. The script-src 'self' 'nonce-PAk3kslfKFAoaP423' allows script content from external scripts loaded from the same origin, or any script element with the specified nonce set as an attribute, e.g., <script nonce='PAk3kslfKFAoaP423'>. Since the attack XSS is an inline script without the specified nonce present, the attack is blocked.

        To get full credit, the answer must mention the nonce, and must not say that the CSP rejects all inline scripts (as inline scripts with the correct nonce still work).

12. Explain why including `'unsafe-inline'` in a CSP makes it almost entirely ineffective at preventing XSS attacks.

    12. The 'unsafe-inline' directive allows any inline <script> to execute, with no restrictions. This means that an attacker-inserted <script> will run.

13. Describe a server-side defense that mitigates the effects of brute force (testing multiple passwords from a dictionary against a single account), credential stuffing (testing username/password pairs obtained from a breach), as well as password spraying (testing a single weak password against a large number of different accounts).

13. There are many valid approaches. For example: rate limiting login attempts based on IP address, rate limited login attempts based on number of distinct accounts particular IP is logging into, showing a CAPTCHA after a certain number of failed attempts, ban users after a certain number of login attempts, deploy multi-factor authentication.

14. Suppose that a page loaded over HTTP loads a login iframe as:

```
<iframe src='https://site.com/login'></iframe>
```

Can an active network attacker steal the password entered into the login frame? Justify your answer.

14. Yes, an active network attacker can modify the top-level page (since it is loaded over HTTP) and replace the iframe with a fake login form which sends the password to the attacker.

15. Suppose an attacker steals the private key of a website that uses TLS, and remains undetected. What can the attacker do using the private key?

15. Man-in-the-middle attack. Note here that the website's **certificate** is already publicly available as part of the TLS protocol, and combined with the stolen **private key** allows the attacker to impersonate the website and/or eavesdrop on supposedly-secure communication.

16. List **three** unique attributes of a user's browser that a fingerprinting script could use to persistently identify the user even if they clear their cookies and other site data.

16. The browser dimensions, the list of installed fonts, the user agent of the browser in use, the specific quirks of their graphics card (canvas fingerprinting), the specific quirks of their audio hardware (web audio fingerprinting), installed browser plugins, color depth, whether the Do Not Track header is sent (ironic)

17. Explain why the browser must send an OPTIONS or "preflight" request to the server before it sends certain HTTP requests. What would happen if the browser didn't send these requests? To help jog your memory, here is an example of an OPTIONS request:

```
OPTIONS /resource/foo
Access-Control-Request-Method: PUT
Origin: https://example.com
```

17. A CORS preflight request is sent so the browser can **check to see if a server understands** the CORS protocol and is okay with the browser issuing potentially-destructive requests.

18. Assume that Axess has an API endpoint `GET /api/transcript` which returns the currently logged-in user's grades. Would the following code running on `https://attacker.com` be allowed to print out your Stanford grades? Justify your answer.

```
<script>
  const res = await fetch('https://axess.stanford.edu/api/transcript')
  const data = await res.body.text()
  console.log(data) // Haha, got your grades!
</script>
```

You can assume that `https://axess.stanford.edu` does **not** send any special HTTP headers such as `Access-Control-Allow-Origin` (also known as "CORS" headers) or set any special cookie attributes such as `SameSite`.

18. No. The request to https://axess.stanford.edu will be sent to the server but **the response will not be readable** by the page because it is a **cross-origin read** which is not allowed **unless there is an Access-Control-Allow-Origin header** present on the response.

19. Would the following code running on `https://attacker.com` be allowed to listen to the 'submit' event on `bank.com`'s login form and grab the username and password? Justify your answer.

```
<iframe src='https://bank.com'></iframe>
<script>
  const loginForm = window.frames[0].forms[0]
  loginForm.addEventListener('submit', () => {
    console.log(loginForm.username) // Haha, got your username...
    console.log(loginForm.password) // ...and password!
  })
</script>
```

19. No. Since https://bank.com and https://attacker.com are different origins, they are not allowed to directly access each other's DOMs across an <iframe> boundary as the attacker's code attempts to do.

20. "Mixing program control and user data" is a class of vulnerabilities where an app accidentally treats user input as code and executes it. Which of the following attacks exploit this class of vulnerabilities? Mark ALL that apply.

- ○ Clickjacking
- ○ CSRF
- ☑ SQL injection
- ○ Denial of service
- ☑ Stored XSS
- ☑ Reflected XSS
- ○ Man-in-the-middle
- ○ None of the above

21. Which of the following attacks might allow an attacker to steal one of your browser cookies?

- ☑ TLS/SSL Strip
- ○ CSRF
- ○ SQL injection
- ☑ JavaScript sandbox escape (compromised renderer process)
- ☑ Stored XSS
- ☑ Reflected XSS
- ○ Brute Force
- ○ None of the above

Problem 3. The Great Cannon (24 points)

In 2015, Github experienced a DoS attack orchestrated by China using the so-called “Great Cannon” (GC). It worked as follows. (Some details have been simplified or modified for this problem.)

Many websites include a fetch for a script for analytics from Baidu, a large Internet service in China somewhat similar to Google. The script would be retrieved via `http://hm.baidu.com/h.js`. The GC operated at the border between China and the rest of the Internet. Upon seeing a request for this script, the GC would prevent the original HTTP request from being forwarded, and would instead return a different script, which instructed clients to repeatedly load `http://github.com/cn-nytimes`, in an attempt to overload Github's servers.

You can assume that Baidu served its traffic using servers in China; Github did so from servers in the USA; and websites using the analytics script were hosted all over the world.

(a) (3 point) Whose traffic contributed to the DDOS attack? **Mark the BEST choice.**

- ○ Web browsers inside China
- ✓ Web browsers outside China
- ○ Both of these
- ○ Neither of these

(b) (6 points) Why doesn't the Same Origin Policy prevent this attack? (2 sentences max)

b. Websites that included the Baidu script were allowed to make requests to GitHub because the **same origin policy allows “simple” GET requests** to be sent to any origin.
Also acceptable: the browser can’t tell a man-in-the-middle attack has happened because the request uses HTTP. Same-origin policy does not apply

(c) (3 points) Which of the following changes would have prevented the attack? Consider each choice in isolation (i.e., assess whether it prevents the attack assuming none of the other choices are in effect). For each choice, assume that the content that the site serves remains the same. **Mark ALL that apply.**

- ✓ Every website that uses Baidu’s analytics changes the script tag URL so it loads over HTTPS instead of HTTP. (Assume the script was also available over HTTPS.)
- ○ Baidu’s analytics server redirects any incoming HTTP connection to a corresponding HTTPS URL.
- ✓ Baidu adds `baidu.com` to the HSTS (HTTP Strict Transport Security) preload list.
- ✓ Baidu switches its analytics server to only be accessible using HTTPS.
- ○ Github’s server redirects any incoming HTTP connection to a corresponding HTTPS URL.
- ○ Github switches its server to only be accessible using HTTPS.
- ○ None of these.

(d) (3 points) Which of the following techniques could Github have used to make the DoS attack ineffective? **Mark ALL that apply.**

- ○ Block any packets from Chinese IP addresses
- ○ Move the affected Github server to a new IP address
- ○ Remove all use of Baidu analytics from Github web pages
- ☑ None of these.

(e) (3 points) The remainder of this problem concerns a Web security feature called Subresource Integrity (SRI). It works by adding an attribute to the script tag for externally loaded scripts:

```
<script src="http://example.com/script.js" integrity="[CRYPTOGOOP]">
```

Browsers then validate the integrity of the script retrieved from the given `src` location.

What should *CRYPTOGOOP* contain for it to achieve its goal of assuring integrity, while minimizing the effort required by web developers to adopt it? **Mark the BEST answer.**

- ○ An encryption of the script being loaded
- ☑ A hash of the script being loaded
- ○ A digital signature of the script being loaded
- ○ A hash of the URL of the script

(f) (3 points) Suppose every website with Baidu's analytics starts using SRI. Given GC's capabilities, could it still redirect some Baidu analytics traffic to Github? Justify your answer.

f. Yes, instead of returning a malicious script (which would fail the integrity check), they can 302 redirect the requests for the analytics scripts to a page on github.com. Requests will be sent to GitHub but the integrity check will fail.

Another answer: Yes, the analytics script is bound to send some requests to Baidu's servers. GC can redirect those requests too.

(g) (3 points) Name **ONE** drawback to a website's owner from deploying SRI. (If you name more than one, we will only grade the first.)

g. Whenever the third-party analytics script changes, the integrity check will fail and the website owner will need to update the integrity hash for it to start working again.

Problem 4. Cookies (9 points)

Your friend has built a personal site hosted at `https://stanford.edu/~clueless`. They have built an authentication system so certain pages of the site can only be accessed by authorized individuals. Once a user logs in successfully, the server sends a response with a `Set-Cookie` HTTP header to set a `sessionId` cookie in the user's browser.

```
Set-Cookie: sessionId=1234; Path=/~clueless
```

Your friend is specifying the `Path` attribute on the cookie so that the cookie is scoped to the path prefix `/~clueless`. This means that the cookie will be sent when the user visits `https://stanford.edu/~clueless` or `https://stanford.edu/~clueless/secret` but not when they visit `https://stanford.edu/~attacker`.

(a) (3 points) Nonetheless, it turns out that `https://stanford.edu/~attacker` can read the `sessionId` cookie that was scoped to your friend's site with the `Path` attribute. Explain what the page at `https://stanford.edu/~attacker` could do to read the cookie.

a. The “Path” attribute does not protect against unauthorized reading of the cookie by other pages on the same origin. So it is possible for https://stanford.edu/~attacker to **include an <iframe> that loads https://stanford.edu/~clueless and then read out the cookie from the frame**. Since both pages are on the same origin, it’s possible for the attacker page to access the DOM of the victim page and read the iframe.contentDocument.cookie property to steal the sessionId cookie.

(b) (3 points) What cookie attribute (e.g. `Secure`, `HttpOnly`, `Domain`, `SameSite`, etc.) could your friend have specified when setting the cookie that would have prevented the attacker from stealing the `sessionId` cookie? Justify your answer.

b. **HttpOnly** would have **prevented the cookie from being accessible to client-side JavaScript**, thus protecting it from pages on the same site.

(c) (3 points) Does adding the cookie attribute you specified in (b) actually prevent `https://stanford.edu/~attacker` from reading the *content* of your friend's website? If yes, explain why. If not, explain how the attacker site can still access the content.

c. **Not protected**. The attacker can still **iframe the https://stanford.edu/~clueless page and reach into the DOM** to read out the private content for a logged in user.

Problem 5. Coffee Shop Wi-Fi (12 points)

You're sitting in a coffee shop enjoying a latte and doing some relaxing computer security reading at `http://awesome-security-stuff.com`. You're connected on the coffee shop's wifi network.

(a) (3 points) Assuming you are only browsing `http://awesome-security-stuff.com`, who is potentially able to observe what articles you are reading? **Mark ALL that apply.**

- [x] Other coffee shop patrons
- [x] The manager of the store next door to the coffee shop who occasionally leeches off of the coffee shop's wifi
- [ ] Your friend in a dorm a few miles away
- [x] The coffee shop's ISP
- [x] The website `awesome-security-stuff.com`
- [ ] None of the above

(b) (3 points) Name a technology that could reduce the number of parties in part (a) that can observe your traffic. Do not give an explanation, simply write down the name.

HTTPS, TLS, VPN

(c) (3 points) If you use the technology you listed in part (b), who will still be able to know a complete list of all the articles you view? **Mark ALL that apply.**

- [ ] Other coffee shop patrons
- [ ] The manager of the store next door to the coffee shop who occasionally leeches off of the coffee shop's wifi
- [ ] Your friend in a dorm a few miles away
- [ ] The coffee shop's ISP
- [x] The website `awesome-security-stuff.com`
- [ ] None of the above

(d) (3 points) You notice that each article has a Facebook Like button, loaded as such:

```
<a href='https://facebook.com/like?url=PAGE_URL'>
  <img src='https://facebook.com/like-button.png' />
</a>
```

allowing you to indicate on Facebook that you enjoyed this article. If Facebook wanted to, could it track what articles you are visiting, if you don't click on the Like button? Justify your answer.

d. Yes. Facebook can inspect the Referer header on the request generated by the <img> element.

Problem 6. Delete Account (6 points)

The route handler below implements the "delete account" functionality which is common on most websites. This allows the user to completely delete their account. The actual deletion logic is in the `deleteAccount` function, which is not shown here. Assume that the session cookie is not a SameSite cookie, i.e. it is set with the `SameSite=None` attribute.

```
app.post('/delete-account', (req, res) => {
  const { sessionId } = req.cookies

  if (!sessionId) {
    // The user is logged out so send them to login page
    res.redirect('/login')
    return
  }

  const { username } = getUserForSessionId(sessionId)
  deleteAccount(username)
  res.send('Account deleted.')
})
```

a. (3 points) There is a severe security issue in the route handler. Identify the issue.

a. **CSRF**. Any website can send a **cross-origin POST** request by **submitting a form** to the /delete-account endpoint, which will delete the currently logged-in user without any user interaction or confirmation.

b. (3 points) Propose a solution that fixes the security issue. Make sure to explain how your proposal actually solves the problem.

b. Any CSRF mitigation would work. Examples:
   - SameSite cookies
   - CSRF authenticity token
   - Require that the user submits their username or password in the request (not as cookie!), which the attacker wouldn't know.
   - Use a non-simple request (e.g., DELETE method)

Problem 7. User Agent (6 points)

The Express below implements a simple website that shows the visitor their browser User Agent. The server also has a feature to display the last 100 user agents that were observed by the server.

```
// Top-level array persists between requests
const userAgents = []

app.get('/', (req, res) => {
  userAgents.push(req.headers['user-agent']) // Save the current user agent

  if (userAgents.length > 100) userAgents.shift() // Only keep latest user agents

  res.send(`
    <h1>Your user agent: ${req.headers['user-agent']}</h1>
    <h1>Last 100 observed user agents:</h1>
    <ul>
      ${userAgents.map(userAgent => `<li>${userAgent}</li>`).join('\n')}
    </ul>
    <p>We take your security seriously, so we use the best encryptions!!!1</p>
  `)
})
```

Recall that the `req.headers` property in Express is an object containing a property for each header in the HTTP request. For example, if the user visits from a Firefox browser, the value of `req.headers['user-agent']` will be `'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:94.0) Gecko/20100101 Firefox/94.0'`.

(a) (3 points) What type of vulnerability does this cause? **Mark the BEST answer.**

- ○ Man-in-the-middle
- ○ CSRF
- ○ Reflected XSS
- ○ SQL injection
- ☑ Stored XSS
- ○ Clickjacking

(b) (3 points) Describe how an attacker could exploit this vulnerability. Include ALL the steps, starting from the steps the attacker performs until the exploit runs.

b. The attacker can curl the web server with a malicious user-agent string (e.g., “<script>alert(document.cookie)</script>”). The server stores this in the userAgents array and includes it in future page loads. The next visitor arrives at the site and the server includes the unsanitized string in the HTML output.

Problem 8. Content Security Policy (9 points)

The following CSP is applied to the given HTML page.

CSP:

```
Content-Security-Policy: default-src 'self'; script-src 'self'; img-src 'self'
https://images.example.com; style-src 'self' https://stylish.example.com;
```

HTML:

```
<!doctype html>
<html lang='en'>
  <head>
    <link rel='stylesheet' href='/style.css' />                              (1)
    <link rel='stylesheet' href='https://stylish.example.com/style.css' />   (2)
  </head>
  <body>
    <script>alert('We have only the BEST memes!')</script>                   (3)

    <h1>Top memes:</h1>
    <img src='https://images.example.com/cat1.jpg'>                          (4)
    <img src='https://images.example.org/cat2.jpg'>                          (5)
    <img src='/memes/cat3.jpg'>                                              (6)

    <script src='/bundle.js'></script>                                       (7)
    <script src='https://partner.example.com/analytics.js'></script>         (8)
  </body>
</html>
```

Specify which resources will be **blocked** from loading by the CSP. There may be more than one. **Mark ALL that apply.**

- ○ 1
- ○ 2
- ✓ ○ 3
- ○ 4
- ○ None will be blocked
- ✓ ○ 5
- ○ 6
- ○ 7
- ✓ ○ 8

Problem 9. Cross Site Script Inclusion (XSSI) (12 points)

In this problem we look at a common Web vulnerability. Consider a banking web site `bank.com`. After login the user is taken to a user information page:

```
https://bank.com/accountInfo.html
```

that shows the user's account balances. `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="https://bank.com/userdata.js">                    (1)
```

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

```
displayData({ name: 'Feross', accountNumber: 1337, balance: 42 })
```

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. Keep in mind that line (1) causes the script `userdata.js` to be executed in the context of the page that includes it.

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. Assume that no special cookie attributes such as `Secure`, `HttpOnly`, or `SameSite` are set.

(a) (3 points) Consider a user who logs into their account at `bank.com` and then visits the URL `https://attacker.com`. Explain how the page at `attacker.com` can cause all of the user's data to be sent to evil.com. Please provide the code contained in the page at `attacker.com`.

a. The attacker site can **define a displayData() function** and then **include the script** from (1) in their page. The user's cookies will be sent with the request and the script will run in the context of attacker.com, calling the attacker's function with the user's account info.
<script>function displayData() { … }</script>
<script src="https://bank.com/userdata.js"></script>

(b) (3 points) How would you keep `accountInfo.html` as a static page, but prevent the attack from part (a)? You need only change line (1) 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 `attacker.com` access. In particular, `userdata.js` need not be a JavaScript file.

b. Change the script to make a **fetch** request for a **JSON** file. Change userdata.js to be a JSON file, so that only websites which are same-origin to bank.com can read the response.

(c) (3 points) Rather than implementing the fix in (b), what cookie attribute could `bank.com` have set on their session cookies to mitigate the issue? **Mark the BEST choice.**

- ○ Secure
- ○ Domain
- ○ HttpOnly
- ☑ SameSite
- ○ Path
- ○ None of the above.

(d) (3 points) What's something you want to do in the new year that you've never done before?

Thank you for an excellent quarter!

Have an ✨ amazing ✨ winter break!