

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".

- _____ 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.

- _____ 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.

- _____ 3. Checking the Referer header is a robust defense against CSRF attacks.

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

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

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

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

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

- _____ 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.

- _____ 10. The design of HTTPS prevents a network attacker from replaying an old session.
- _____ 11. The cookie attribute `HttpOnly` helps to mitigate the effects of XSS attacks by preventing client-side JavaScript from reading the cookie.
- _____ 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.
- _____ 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`.
- _____ 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.
- _____ 15. The best way to prevent untrusted user input from exploiting your application is to use encryption.
- _____ 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.
- _____ 17. When a site sends a **cross-origin** GET request **without** custom headers, the browser will first send a "preflight" OPTIONS request.
- _____ 18. When a site sends a **same-origin** PUT request **with** custom headers, the browser will first send a "preflight" OPTIONS request.

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

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.

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.

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>

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-PAk3ks1fKFAoaP423';

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

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

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).

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.

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?

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.

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
```

18. Assume that Axxess 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.

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>
```

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.

- | | |
|---|---|
| <input type="radio"/> Clickjacking | <input type="radio"/> Stored XSS |
| <input type="radio"/> CSRF | <input type="radio"/> Reflected XSS |
| <input type="radio"/> SQL injection | <input type="radio"/> Man-in-the-middle |
| <input type="radio"/> Denial of service | <input type="radio"/> None of the above |

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

- | | |
|---|---|
| <input type="radio"/> TLS/SSL Strip | <input type="radio"/> Stored XSS |
| <input type="radio"/> CSRF | <input type="radio"/> Reflected XSS |
| <input type="radio"/> SQL injection | <input type="radio"/> Brute Force |
| <input type="radio"/> JavaScript sandbox escape
(compromised renderer process) | <input type="radio"/> 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)

(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
- Remove all use of Baidu analytics from Github web pages
- Move the affected Github server to a new IP address
- 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 digital signature of the script being loaded
- A hash 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.

(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.)

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.
- (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.
- (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.

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.**

- Other coffee shop patrons
- The coffee shop's ISP
- The manager of the store next door to the coffee shop who occasionally leeches off of the coffee shop's wifi
- The website awesome-security-stuff.com
- Your friend in a dorm a few miles away
- 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.

(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 coffee shop's ISP
- The manager of the store next door to the coffee shop who occasionally leeches off of the coffee shop's wifi
- The website awesome-security-stuff.com
- Your friend in a dorm a few miles away
- 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.

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.

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

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.**

- | | |
|---|-------------------------------------|
| <input type="radio"/> Man-in-the-middle | <input type="radio"/> CSRF |
| <input type="radio"/> Reflected XSS | <input type="radio"/> SQL injection |
| <input type="radio"/> Stored XSS | <input type="radio"/> 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.

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.**

- | | |
|--|-------------------------|
| <input type="radio"/> 1 | <input type="radio"/> 5 |
| <input type="radio"/> 2 | <input type="radio"/> 6 |
| <input type="radio"/> 3 | <input type="radio"/> 7 |
| <input type="radio"/> 4 | <input type="radio"/> 8 |
| <input type="radio"/> None will be blocked | |

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`.

(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.

(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!