

Stanford University
Computer Science Department
CS 253 Final Exam Fall 2019

December 10, 2019

This is a closed book exam. You may use two (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
True/False (14 points)	
Short Answer (22 points)	
Free Response (37 points)	
TOTAL (73 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:

True or False (1 point each) –

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

1. Code injection is caused when untrusted user data unexpectedly becomes code.
2. It's OK to put sensitive information in hidden form fields since, after all, they're hidden. For example, `<input type='hidden' name='databasePassword' value='hunter2'>`.
3. The server can trust cookie values in HTTP requests to be untampered since the cookies are set by the server.
4. The cookie attribute `HttpOnly` helps to mitigate the effects of XSS attacks by preventing client-side JavaScript from reading the cookie.
5. 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)
6. You should prefer to use a blacklist (to block known bad input and allow everything else) rather than an allowlist (to only allow known good input and block everything else).
7. Cross-site request forgery is a type of injection attack.
8. HTML/JavaScript are the primary languages targeted by cross-site scripting attacks.
9. Reflected XSS occurs when a malicious user convinces a victim to send a request to a server with malicious input and the server echoes the input back to client.

10. The best way to prevent untrusted user input from exploiting your application is to use encryption.
11. You should set the Secure flag in a cookie to ensure that the cookie is only sent over encrypted HTTPS connections.
12. 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.
13. Two-factor authentication (a password together with a Time-based One-time Password (TOTP) code) is an example of defense-in-depth.
14. The XSS Auditor was removed from Chrome because an attacker could use it to prevent specific scripts within a targeted page from executing.

Short Answer (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. Which character is most likely to be used in a SQL injection attack? Choose from: the single quote ('), the null byte, the less than sign (<), or the greater than sign (>).

6. An attacker injects an XSS payload into the HTML page sent by your server. Given the following CSP, would the XSS attack succeed? Why or why not?

CSP: Content-Security-Policy: script-src 'self';

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

7. An attacker injects an XSS payload into the HTML page sent by your server. Given the following CSP, would the XSS attack succeed? Why or why not?

CSP: Content-Security-Policy: script-src 'self' https://javascript-cdn.com;

XSS: `<script src='https://javascript-cdn.com/attacker-script.js'></script>`

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

9. Web browsers like Firefox and operating systems like macOS and Windows ship with a large built-in list of public keys of Certificate Authorities. What are these used for?

10. 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).

11. What is the difference between authentication and authorization?

Free Response (3 points each) –

For each question, write an answer using no more than **150 words**.

1. Same Origin Policy:

Would the following code running on <https://attacker.com> be allowed to print out the contents of the Axess homepage, which include the currently logged-in user's grades? Why or why not?

```
<script>
  const res = await fetch('https://axess.stanford.edu')
  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, which are also known as "CORS" headers.

2. More Same Origin Policy:

Would the following code running on <https://attacker.com> be allowed to listen to the 'submit' event on the bank's login form and grab the username and password? Why or why not?

```
<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>
```

3. CORS Preflight:

Explain why the browser must send an OPTIONS or "preflight" request to the server before it sends certain HTTP requests. To help jog your memory, here is an example of an OPTIONS request:

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


4. Cookies:

Your friend has built a personal site hosted at <https://stanford.edu/~victim>. They have built an authentication system so certain pages of the site can only be accessed by specific 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=~victim
```

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

Explain how <https://stanford.edu/~attacker> can nonetheless read the sessionId cookie that was scoped to the victim's site.

5. More Cookies:

An attacker includes the following HTML in their site hosted at <https://attacker.com> which makes a GET request to a vulnerable bank server and transfers money into the attacker's account.

```
<img src='https://bank.com/withdraw?amount=1000&to=attacker' />
```

The attacker is hoping the user is already authenticated with the bank site before they visit <https://attacker.com> and send the above GET request to the bank. The attacker entices users to visit their site by including hundreds of cute kittens like these ones:



Explain how the bank can modify their server code to protect users from this attack.

6. XSS:

The following Express route handler implements the homepage of the site at <https://insecure.example.com> but it is vulnerable to reflected XSS.

```
app.get('/', (req, res) => {
  let welcomeMessage = 'Welcome to our site!'
  if (req.query.source) {
    welcomeMessage = `Welcome ${req.query.source} reader!`
  }

  res.send(`
    <h1>${welcomeMessage}</h1>
    <p>This site uses top-of-the-line security and encryptions!!!1</p>
  `)
})
```

Recall that the `req.query` property in Express is an object containing a property for each query string parameter in the route. For example, if the user visits <https://insecure.example.com/?name=zelda>, then the value of `req.query` will be `{ name: 'zelda' }`. If there is no query string, it is the empty object, `{}`.

Describe the XSS vulnerability in the code and provide a URL which an attacker could get a victim to visit in order to pull off a reflected XSS attack against them. The URL you provide should execute the following code: `alert(document.cookie)`.

7. More XSS:

The following Express route handler implements the logic for the login form of <https://insecure.example.com> but it is vulnerable to reflected XSS.

```
app.post('/login', (req, res) => {
  const { username, password } = req.body
  if (isAuthValid(username, password)) {
    res.send(`
      <h1>Welcome logged in user!</h1>
      <script>
        let username = '${jsStringEscape(username)}'
        alert('Hi there, ' + username)
      </script>
    `)
  } else {
    res.send('Invalid username or password!')
  }
})

// Escape a string so it can safely be used inside a JavaScript string within
// a <script> tag in an HTML page.
function jsStringEscape (str) {
  return str
    .replace(/'/g, "\\'") // Replace all ' with \'
    .replace(/"/g, "\\\"") // Replace all " with \"
}

// Returns true if the given login credentials are valid. False, otherwise.
function isAuthValid (username, password) {
  // implementation omitted...
}
```

Describe the XSS vulnerability in the code and provide a URL which an attacker could get a victim to visit in order to pull off a reflected XSS attack against them. The URL you provide should execute the following code: `alert(document.cookie)`.

Hint: Take a close look at the `jsStringEscape` function – it doesn't escape all the necessary characters!

Another hint: Think about what the final HTML page will look like with different possible username values!

Another, another hint: There are actually two vulnerabilities, but you only need to find one of them.

(Continued...) 7. More XSS:

8. CSP:

The given CSP is applied to the given HTML page. Specify which resources, if any, will be *blocked* from loading by the CSP. There may be more than one.

```
CSP: Content-Security-Policy: default-src 'none'; script-src 'self'
https://partner.example.com; img-src 'self' https://images.example.com; style-src
'self';
```

HTML:

```
<!doctype html>
<html lang='en'>
  <head>
    <link rel='stylesheet' href='/style.css' />
    <link rel='stylesheet' href='https://stylish.example.com/style.css' />
  </head>
  <body>
    <script>alert('We have only the BEST memes!')</script>
    <h1>Top memes:</h1>
    <img src='https://images.example.com/cat1.jpg'>
    <img src='https://images.example.com/cat2.jpg'>
    <img src='/memes/cat3.jpg'>
    <script src='/bundle.js'></script>
    <script src='https://partner.example.com/analytics.js'></script>
  </body>
</html>
```

9. HSTS Preload:

An attacker performed a DNS hijacking attack against your domain name. The attacker changed your domain's A record to point to their server IP address instead of yours. (Web browsers use the DNS A record to translate domain names to IP addresses). With the attacker in control of the DNS responses that your site visitors receive, their browsers will be directed to connect to the attacker's server instead of yours. Fortunately, your site is served using TLS and your site is loaded into the HTTPS Strict Transport Security (HSTS) Preload List.

Explain how TLS and HSTS Preload protects visitors to your site from this attack.

10. Command injection:

The following Node.js program implements an HTTP server which accepts a user-provided filename and returns the contents of the specified file to the user, if it exists on the server. The file should only be returned if it exists in a folder named "static" where static files intended for viewing are stored.

```
const express = require('express')
const childProcess = require('child_process')
const app = express()

app.get('/', (req, res) => {
  res.send(`
    <h1>File viewer</h1>
    <form method='GET' action='/view'>
      <input name='filename' />
      <input type='submit' value='Submit' />
    </form>
  `)
})

app.get('/view', (req, res) => {
  const { filename } = req.query

  try {
    const stdout = childProcess.execSync('cat static/' + filename)
    // command succeeded, file exists
    res.send(stdout.toString())
  } catch (err) {
    // command failed, file does not exist
    res.send(err.toString())
  }
})

app.listen(4000, '127.0.0.1')
```

Recall, the `execSync` function takes one or more commands to run, and runs them. If the command succeeds, the function returns the standard output. Otherwise, it throws an exception.

Also recall, the `cat` program reads files sequentially, writing them to standard output. For example, the command `cat file.txt` will cause the contents of `file.txt` to be printed to the terminal.

Here's an example request and response interaction with this server.

(Continued on next page...)

(Continued...) 10. Command injection

Request:

```
GET /view?filename=hello.txt HTTP/1.1
Host: localhost:4000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_1)
```

Response:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Date: Tue, 10 Dec 2019 00:00:00 GMT
```

Hello, world!

There is a glaring security vulnerability in this server. What is the issue? How could the issue be fixed?

Hint: There are actually two security vulnerabilities, but you only need to find one of them.

11. Fingerprinting:

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.

12. Logic bug:

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. To confirm that the request came from the actual user, the request must include the user's password which is validated before the account is deleted.

```
app.get('/delete', (req, res) => {
  const { username, password } = req.body

  if (!isAuthValid(username, password)) {
    res.send('Invalid username or password.')
  }

  deleteAccount(username)

  res.send('Account deleted.')
})
```

There are two severe security issues in the route handler. Identify the two issues.

(Continued...) 12. Logic bug:

13. Winter break (1 point):

What are you most looking forward to doing during the winter break?

Have an amazing winter break!