# PortSwigger Web Cache Deception Lab Notes

**Web Cache Deception (WCD)** is a web attack where an attacker tricks a caching server (like a CDN or reverse proxy) into caching **sensitive, user-specific content** (e.g., account info, session data) under a **publicly accessible URL**.

Because caches are meant to store and serve responses to many users, if sensitive data is cached due to poor cache-control rules or URL manipulation (like adding /style.css after a personal URL), the attacker or any other user can later request that cached URL and receive **private information belonging to someone else**.

👉 In short: it abuses misconfigured caching to turn **personal responses into publicly cacheable pages**.

1. Exploiting path mapping for web cache deception

To solve the lab, find the API key for the user carlos. You can log in to your own account using the following credentials: wiener:peter.

**Required knowledge**

To solve this lab, you'll need to know:

* How regex endpoints map URL paths to resources.
* How to detect and exploit discrepancies in the way the cache and origin server map URL paths.

Solution

**Identify a target endpoint**

1. In Burp's browser, log in to the application using the credentials wiener:peter.
2. Notice that the response contains your API key.

**Identify a path mapping discrepancy**

1. In **Proxy > HTTP history**, right-click the GET /my-account request and select **Send to Repeater**.
2. Go to the **Repeater** tab. Add an arbitrary segment to the base path, for example change the path to /my-account/abc.
3. Send the request. Notice that you still receive a response containing your API key. This indicates that the origin server abstracts the URL path to /my-account.
4. Add a static extension to the URL path, for example /my-account/abc.js.
5. Send the request. Notice that the response contains the X-Cache: miss and Cache-Control: max-age=30 headers. The X-Cache: miss header indicates that this response wasn't served from the cache. The Cache-Control: max-age=30 header suggests that if the response has been cached, it should be stored for 30 seconds.
6. Resend the request within 30 seconds. Notice that the value of the X-Cache header changes to hit. This shows that it was served from the cache. From this, we can infer that the cache interprets the URL path as /my-account/abc.js and has a cache rule based on the .js static extension. You can use this payload for an exploit.

**Craft an exploit**

1. In Burp's browser, click **Go to exploit server**.
2. In the **Body** section, craft an exploit that navigates the victim user carlos to the malicious URL that you crafted earlier. Make sure to change the arbitrary path segment you added, so the victim doesn't receive your previously cached response:

<script>document.location="https://YOUR-LAB-ID.web-security-academy.net/my-account/wcd.js"</script>

1. Click **Deliver exploit to victim**. When the victim views the exploit, the response they receive is stored in the cache.
2. Go to the URL that you delivered to carlos in your exploit:

https://YOUR-LAB-ID.web-security-academy.net/my-account/wcd.js

1. Notice that the response includes the API key for carlos. Copy this.
2. Click **Submit solution**, then submit the API key for carlos to solve the lab.
3. Exploiting path delimiters for web cache deception

To solve the lab, find the API key for the user carlos. You can log in to your own account using the following credentials: wiener:peter.

We have provided a list of possible delimiter characters to help you solve the lab: [Web cache deception lab delimiter list](https://portswigger.net/web-security/web-cache-deception/wcd-lab-delimiter-list).

**Required knowledge**

To solve this lab, you'll need to know:

* How to identify discrepancies in how the cache and origin server interpret characters as delimiters.
* How delimiter discrepancies can be used to exploit a static directory cache rule.

Solution

**Identify a target endpoint**

1. In Burp's browser, log in to the application using the credentials wiener:peter.
2. Notice that the response contains your API key.

**Identify path delimiters used by the origin server**

1. In **Proxy > HTTP history**, right-click the GET /my-account request and select **Send to Repeater**.
2. Go to the **Repeater** tab. Add an arbitrary segment to the path. For example, change the path to /my-account/abc.
3. Send the request. Notice the 404 Not Found response with no evidence of caching. This indicates that the origin server doesn't abstract the path to /my-account.
4. Remove the arbitrary segment and add an arbitrary string to the original path. For example, change the path to /my-accountabc.
5. Send the request. Notice the 404 Not Found response with no evidence that the response was cached. You'll use this response as a reference to help you identify characters that aren't used as delimiters.
6. Right-click the request and select **Send to Intruder**.
7. Go to the **Intruder** tab. Make sure that **Sniper attack** is selected and add a payload position after /my-account as follows: /my-account§§abc.
8. In the **Payloads** side panel, under **Payload configuration**, add a list of characters that may be used as delimiters.
9. Under **Payload encoding**, deselect **URL-encode these characters**.
10. Click **Start attack**. The attack runs in a new window.
11. When the attack finishes, sort the results by **Status code**. Notice that the ; and ? characters receive a 200 response with your API key. All other characters receive the 404 Not Found response. This indicates that the origin server uses ; and ? as path delimiters.

**Investigate path delimiter discrepancies**

1. Go to the **Repeater** tab that contains the /my-accountabc request.
2. Add the ? character after /my-account and add a static file extension to the path. For example, update the path to /my-account?abc.js.
3. Send the request. Notice that the response doesn't contain evidence of caching. This may indicate that the cache also uses ? as a path delimiter.
4. Repeat this test using the ; character instead of ?. Notice that the response contains the X-Cache: miss header.
5. Resend the request. Notice that the value of the X-Cache header changes to hit. This indicates that the cache doesn't use ; as a path delimiter and has a cache rule based on the .js static extension. You can use this payload for an exploit.

**Craft an exploit**

1. In Burp's browser, click **Go to exploit server**.
2. In the **Body** section, craft an exploit that navigates the victim user carlos to the malicious URL you crafted earlier. Make sure to change the arbitrary string, so the cache creates a unique key and carlos caches their account details instead of receiving your previously cached response:

<script>document.location="https://YOUR-LAB-ID.web-security-academy.net/my-account;wcd.js"</script>

1. Click **Deliver exploit to victim**. When the victim views the exploit, the response they receive is stored in the cache.
2. Go to the URL that you delivered to carlos:

https://YOUR-LAB-ID.web-security-academy.net/my-account;wcd.js

1. Notice that the response includes the API key for carlos. Copy this.
2. Click **Submit solution**, then submit the API key for carlos to solve the lab.
3. Exploiting origin server normalization for web cache deception

To solve the lab, find the API key for the user carlos. You can log in to your own account using the following credentials: wiener:peter.

We have provided a list of possible delimiter characters to help you solve the lab: [Web cache deception lab delimiter list](https://portswigger.net/web-security/web-cache-deception/wcd-lab-delimiter-list).

**Required knowledge**

To solve this lab, you'll need to know:

* How to identify whether the cache and origin server normalize the URL path.
* How to identify static directory cache rules.
* How to exploit normalization by the origin server.

Solution

**Identify a target endpoint**

1. In Burp's browser, log in to the application using the credentials wiener:peter.
2. Notice that the response contains your API key.

**Investigate path delimiter discrepancies**

1. In **Proxy > HTTP history**, right-click the GET /my-account request and select **Send to Repeater**.
2. Go to the **Repeater** tab. Change the path to /my-account/abc, then send the request. Notice the 404 Not Found response. This indicates that the origin server doesn't abstract the path to /my-account.
3. Change the path to /my-accountabc, then send the request. Notice that this returns a 404 Not Found response with no evidence of caching.
4. Right-click the message and select **Send to Intruder**.
5. Go to the **Intruder** tab. Make sure that **Sniper attack** is selected and add a payload position after /my-account as follows: /my-account§§abc.
6. In the **Payloads** side panel, under **Payload configuration**, add a list of characters that may be used as delimiters. Under **Payload encoding**, deselect **URL-encode these characters**.
7. Click **Start attack**. The attack runs in a new window.
8. When the attack finishes, sort the results by **Status code**. Notice that only the ? character receives a 200 response with your API key. This indicates that the origin server only uses ? as a path delimiter. As ? is generally universally used as a path delimiter, move on to investigate normalization discrepancies.

**Investigate normalization discrepancies**

1. In **Repeater**, remove the arbitrary abc string and add an arbitrary directory followed by an encoded dot-segment to the start of the original path. For example, /aaa/..%2fmy-account.
2. Send the request. Notice that this receives a 200 response with your API key. This indicates that the origin server decodes and resolves the dot-segment, interpreting the URL path as /my-account.
3. In **Proxy > HTTP history**, notice that the paths for static resources all start with the directory prefix /resources. Notice that responses to requests with the /resources prefix show evidence of caching.
4. Right-click a request with the prefix /resources and select **Send to Repeater**.
5. In **Repeater**, add an encoded dot-segment after the /resources path prefix, such as /resources/..%2fYOUR-RESOURCE.
6. Send the request. Notice that the 404 response contains the X-Cache: miss header.
7. Resend the request. Notice that the value of the X-Cache header changes to hit. This may indicate that the cache doesn't decode or resolve the dot-segment and has a cache rule based on the /resources prefix. To confirm this, you'll need to conduct further testing. It's still possible that the response is being cached due to a different cache rule.
8. Modify the URL path after /resources to a arbitrary string as follows: /resources/aaa. Send the request. Notice that the 404 response contains the X-Cache: miss header.
9. Resend the request. Notice that the value of the X-Cache header changes to hit. This confirms that there is a static directory cache rule based on the /resources prefix.

**Craft an exploit**

1. Go to the **Repeater** tab that contains the /aaa/..%2fmy-account request. Attempt to construct an exploit as follows: /resources/..%2fmy-account. Send the request. Notice that this receives a 200 response with your API key and the X-Cache: miss header.
2. Resend the request and notice that the value of the X-Cache header updates to hit.
3. In Burp's browser, click **Go to exploit server**.
4. In the **Body** section, craft an exploit that navigates the victim user carlos to a malicious URL. Make sure to add an arbitrary parameter as a cache buster, so the victim doesn't receive your previously cached response:

<script>document.location="https://YOUR-LAB-ID.web-security-academy.net/resources/..%2fmy-account?wcd"</script>

1. Click **Deliver exploit to victim**. When the victim views the exploit, the response they receive is stored in the cache.
2. Go to the URL that you delivered to carlos in your exploit:

https://YOUR-LAB-ID.web-security-academy.net/resources/..%2fmy-account?wcd

1. Notice that the response includes the API key for the user carlos. Copy this.
2. Click **Submit solution**, then submit the API key for carlos to solve the lab.
3. Exploiting cache server normalization for web cache deception

To solve the lab, find the API key for the user carlos. You can log in to your own account using the following credentials: wiener:peter.

We have provided a list of possible delimiter characters to help you solve the lab: [Web cache deception lab delimiter list](https://portswigger.net/web-security/web-cache-deception/wcd-lab-delimiter-list).

**Required knowledge**

To solve this lab, you'll need to know:

* How to identify whether the cache and origin server normalize the URL path.
* How to identify static directory cache rules.
* How to identify discrepancies in how the cache and origin server interpret characters as delimiters.
* How to exploit normalization by the cache server.

Solution

**Identify a target endpoint**

1. In Burp's browser, log in to the application using the credentials wiener:peter.
2. Notice that the response contains your API key.

**Investigate path delimiters used by the origin server**

1. In **Proxy > HTTP history**, right-click the GET /my-account request and select **Send to Repeater**.
2. Change the URL path to /my-account/abc, then send the request. Notice the 404 Not Found response. This indicates that the origin server doesn't abstract the path to /my-account.
3. Change the path to /my-accountabc, then send the request. Notice that this returns a 404 Not Found response with no evidence of caching.
4. Right-click the message and select **Send to Intruder**.
5. Go to the **Intruder** tab. Make sure that **Sniper attack** is selected and add a payload position after /my-account as follows: /my-account§§abc.
6. In the **Payloads** side panel, under **Payload configuration**, add a list of characters that may be used as delimiters.
7. Under **Payload encoding**, deselect **URL-encode these characters**.
8. Click **Start attack**. The attack runs in a new window.
9. When the attack finishes, sort the results by **Status code**. Notice that the #, ?, %23, and %3f characters receive a 200 response with your API key. This indicates that they're used by the origin server as path delimiters. Ignore the # character. It can't be used for an exploit as the victim's browser will use it as a delimiter before forwarding the request to the cache.

**Investigate path delimiter discrepancies**

1. Go to the **Repeater** tab that contains the /my-accountabc request. Add the ? character after /my-account and add a static extension to the path. For example, update the path to /my-account?abc.js.
2. Send the request. Notice that the response doesn't contain evidence of caching. This either indicates that the cache also uses ? as a path delimiter, or that the cache doesn't have a rule based on the .js extension.
3. Repeat this test using the %23 and %3f characters instead of ?. Notice that the responses don't show evidence of caching.

**Investigate normalization discrepancies**

1. Remove the query string and add an arbitrary directory followed by an encoded dot-segment to the start of the original path. For example, /aaa/..%2fmy-account.
2. Send the request. Notice that this receives a 404 response. This indicates that the origin server doesn't decode or resolve the dot-segment to normalize the path to /my-account.
3. In **Proxy > HTTP history**, notice that static resources share the URL path directory prefix /resources. Notice that responses to requests with the /resources prefix show evidence of caching.
4. Right-click a request with the prefix /resources and select **Send to Repeater**.
5. In **Repeater**, add an encoded dot-segment and arbitrary directory before the /resources prefix. For example, /aaa/..%2fresources/YOUR-RESOURCE.
6. Send the request. Notice that the 404 response contains the X-Cache: miss header.
7. Resend the request. Notice that the value of the X-Cache header updates to hit. This may indicate that the cache decodes and resolves the dot-segment and has a cache rule based on the /resources prefix. To confirm this, you'll need to conduct further testing. It's still possible that the response is being cached due to a different cache rule.
8. Add an encoded dot-segment after the /resources path prefix as follows: /resources/..%2fYOUR-RESOURCE.
9. Send the request. Notice that the 404 response no longer contains evidence of caching. This indicates that the cache decodes and resolves the dot-segment and has a cache rule based on the /resources prefix.

**Craft an exploit**

1. Go to the **Repeater** tab that contains the /aaa/..%2fmy-account request. Use the ? delimiter to attempt to construct an exploit as follows:

/my-account?%2f%2e%2e%2fresources

1. Send the request. Notice that this receives a 200 response with your API key, but doesn't contain evidence of caching.
2. Repeat this test using the %23 and %3f characters instead of ?. Notice that when you use the %23 character this receives a 200 response with your API key and the X-Cache: miss header. Resend and notice that this updates to X-Cache: hit. You can use this delimiter for an exploit.
3. In Burp's browser, click **Go to exploit server**.
4. In the **Body** section, craft an exploit that navigates the victim user carlos to a malicious URL. Make sure to add an arbitrary parameter as a cache buster:

<script>document.location="https://YOUR-LAB-ID.web-security-academy.net/my-account%23%2f%2e%2e%2fresources?wcd"</script>

1. Click **Deliver exploit to victim**.
2. Go to the URL that you delivered to carlos in your exploit:

https://YOUR-LAB-ID.web-security-academy.net/my-account%23%2f%2e%2e%2fresources?wcd

1. Notice that the response includes the API key for the user carlos. Copy this.
2. Click **Submit solution**, then submit the API key for carlos to solve the lab.