**Exploiting static directory cache rules:**

**\*\* THIS IS THE PATH TRAVERSAL TYPE CACHE DECEPTION \*\***

**\*\* ONCE AGAIN THIS MUST BE TESTED MANUALLY – the resources directory will be different between sites so its better just to test this manually. \*\***

It's common practice for web servers to store static resources in specific directories. Cache rules often target these directories by matching specific URL path prefixes, like /static, /assets, /scripts, or /images. These rules can also be vulnerable to web cache deception.

**Normalization discrepancies**

Normalization involves converting various representations of URL paths into a standardized format. This sometimes includes decoding encoded characters and resolving dot-segments, but this varies significantly from parser to parser.

Discrepancies in how the cache and origin server normalize the URL can enable an attacker to construct a path traversal payload that is interpreted differently by each parser. Consider the example /static/..%2fprofile:

* An origin server that decodes slash characters and resolves dot-segments would normalize the path to /profile and return profile information.
* A cache that doesn't resolve dot-segments or decode slashes would interpret the path as /static/..%2fprofile. If the cache stores responses for requests with the /static prefix, it would cache and serve the profile information.

As shown in the above example, each dot-segment in the path traversal sequence needs to be encoded. Otherwise, the victim's browser will resolve it before forwarding the request to the cache. Therefore, an exploitable normalization discrepancy requires that either the cache or origin server decodes characters in the path traversal sequence as well as resolving dot-segments.

\*\* From what ive seen, /../ basically always gets resolved \*\*

------------------------------------------------------------

OPTIONAL DETECTION INFORMATION:

**DETECTING NORMALIZATION by Origin Server:**

To test how the origin server normalizes the URL path, send a request to a non-cacheable resource with a path traversal sequence and an arbitrary directory at the start of the path. To choose a non-cacheable resource, look for a non-idempotent method like POST. For example, modify /profile to /aaa/..%2fprofile:

* If the response matches the base response and returns the profile information, this indicates that the path has been interpreted as /profile. The origin server decodes the slash and resolves the dot-segment.
* If the response doesn't match the base response, for example returning a 404 error message, this indicates that the path has been interpreted as /aaa/..%2fprofile. The origin server either doesn't decode the slash or resolve the dot-segment.

**VERY USEFUL TIP:**

**\*\*\* When testing for normalization, start by encoding only the second slash in the dot-segment. This is important because some CDNs match the slash following the static directory prefix. You can also try encoding the full path traversal sequence, or encoding a dot instead of the slash. This can sometimes impact whether the parser decodes the sequence. \*\*\***

------------

**Detecting normalization by the cache server:**

You can use a few different methods to test how the cache normalizes the path. Start by identifying potential static directories. In Proxy > HTTP history, look for requests with common static directory prefixes and cached responses. Focus on static resources by setting the HTTP history filter to only show messages with 2xx responses and script, images, and CSS MIME types.

You can then **choose a request with a cached response and resend the request with a path traversal sequence and an arbitrary directory at the start of the static path**. **Choose a request with a response that contains evidence of being cached.** For example, /aaa/..%2fassets/js/stockCheck.js:

* If the response is no longer cached, this indicates that the cache isn't normalizing the path before mapping it to the endpoint. It shows that there is a cache rule based on the /assets prefix.
* If the response is still cached, this may indicate that the cache has normalized the path to /assets/js/stockCheck.js.

You can also add a path traversal sequence after the directory prefix. For example, modify /assets/js/stockCheck.js to /assets/..%2fjs/stockCheck.js:

* If the response is no longer cached, this indicates that the cache decodes the slash and resolves the dot-segment during normalization, interpreting the path as /js/stockCheck.js. It shows that there is a cache rule based on the /assets prefix.
* If the response is still cached, this may indicate that the cache hasn't decoded the slash or resolved the dot-segment, interpreting the path as /assets/..%2fjs/stockCheck.js.

Note that in both cases, the response may be cached due to another cache rule, such as one based on the file extension. To confirm that the cache rule is based on the static directory, replace the path after the directory prefix with an arbitrary string. For example, /assets/aaa. If the response is still cached, this confirms the cache rule is based on the /assets prefix. Note that if the response doesn't appear to be cached, this doesn't necessarily rule out a static directory cache rule as sometimes 404 responses aren't cached. \*\* Maybe try /assets/js/ if we get an uncached 404.

**It's possible that you may not be able to definitively determine whether the cache decodes dot segments and decodes the URL path without attempting an exploit. Basically, just begin testing for vulnerabilities. Don’t look for 404, just try to access the data.**

------------------------------------------------------------

**Exploiting Normalization by Origin Server:**

If the origin server resolves encoded dot-segments, but the cache doesn't, you can attempt to exploit the discrepancy by constructing a payload according to the following structure:

/<static-directory-prefix>/..%2f<dynamic-path>

For example, consider the payload /assets/..%2fprofile:

* The cache interprets the path as: /assets/..%2fprofile
* The origin server interprets the path as: /profile

The origin server returns the dynamic profile information, which is stored in the cache.

**\*\*\*\* True process for exploitation: craft the URL payload and add a cache buster to the end of the sensitive endpoint eg /assets/..%2fprofile?123 (this will ensure we don’t see our previously cached response). Visit the URL with the victim account and GET IT CACHED. Then request that same URL payload (/assets/..%2fprofile?123 ) as a different or unauthenticated user and observe that we see the victim’s information \*\*\***

**------------**

**Exploiting normalization by Cache Server:**

**\*\* This is essentially the opposite as we do /sensitive %2f%2e%2e%2fstatic-resource-dir HOWEVER: we will also need to identify a delimiter to properly exploit this detailed below:**

If the cache server resolves encoded dot-segments but the origin server doesn't, you can attempt to exploit the discrepancy by constructing a payload according to the following structure:

/<dynamic-path>%2f%2e%2e%2f<static-directory-prefix>

**\*\*NOTE: When exploiting normalization by the cache server, encode all characters in the path traversal sequence. Using encoded characters helps avoid unexpected behavior when using delimiters, and there's no need to have an unencoded slash following the static directory prefix since the cache will handle the decoding.\*\* THIS MUST BE TESTED MANUALLY!!**

In this situation, path traversal alone isn't sufficient for an exploit. For example, consider how the cache and origin server interpret the payload /profile%2f%2e%2e%2fstatic:

* The cache interprets the path as: /static
* The origin server interprets the path as: /profile%2f%2e%2e%2fstatic

The origin server is likely to return an error message instead of profile information.

To exploit this discrepancy, you'll need to also identify a delimiter that is used by the origin server but not the cache. Test possible delimiters by adding them to the payload after the dynamic path:

* If the origin server uses a delimiter, it will truncate the URL path and return the dynamic information.
* If the cache doesn't use the delimiter, it will resolve the path and cache the response.

For example, consider the payload /profile;%2f%2e%2e%2fstatic. The origin server uses ; as a delimiter:

* The cache interprets the path as: /static
* The origin server interprets the path as: /profile

The origin server returns the dynamic profile information, which is stored in the cache. You can therefore use this payload for an exploit.

**\*\* this will likely require fuzzing. We want to find a delimiter that still results in a 200 ok (as opposed to a 404). Be sure to test all encodings as well. Almost always the cache server will decode stuff but the backend will not meaning that we will often HAVE to encode the delimiter just as we need to encode the path traversal in order to exploit \*\***