

# Protecting transfer data in Documentum REST Services

---

Web security is a field that includes many threat models and counter-attack technologies. In Documentum REST Services release 7.3, we provide users with a number of web security related configuration parameters that help to protect the transfer data in the REST API. In this post, we will talk about what they are, and how to configure them.

The security models covered in this post include:

- **Cross-site request forgery (CSRF)**
- **Cross-origin resource sharing (CORS)**
- **Request sanitization**
- **HTTP strict transport security (HSTS)**
- **Cacheable HTTPS response**
- **Browser cross-site scripting (XSS)**
- **Cross-frame scripting (XFS)**
- **Response content sniffing**

## Cross-site request forgery (CSRF)

[Cross-Site Request Forgery \(CSRF\)](#) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. CSRF attacks specifically target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request. Here is a simple scenario.

1. Alice logs in to her bank account via web browser and the browser remembers a session cookie (doesn't logout)
2. Alice visits another website that tells her she can earn some money for free

3. Alice clicks below form showing "**Win Money!**"

```
<form action="https://bank.example.com/transfer.json" method="post" enctype="text/plain"> <input name='{ "amount":100,"routingN
```

4. The request is actually redirected to her bank website with a POST method to transfer her money out from her account.

5. Alice has no perception on the money lost because she was not asked for a second login.

In Documentum REST Services, we support **Client Token** cookie authentication for some specific authentication schemes, for instance, Central Authentication Services (CAS), SAML2 Web SSO and so on. The Client Token cookie can be remembered by web browsers until it times out. The above vulnerable user scenario won't occur in Documentum REST Services because Documentum REST server requires a vendor specific content type **application/vnd.emc.documentum+xml** or **application/vnd.emc.documentum+json** for any HTTP POST methods. However, as HTTP cookie is used, the weakness of the cookie preservation still exists. To protect Client Token cookie against CSRF attack, we enable CSRF protection by default in Documentum REST 7.3.

## Client Token authentication with CSRF token

In REST server runtime properties file, there is a parameter to turn on/off CSRF protection for Client Token cookie. The value is true by default, meaning CSRF token is required.

```
rest.security.csrf.enabled= true | false
```

When it is enabled, the Client Token authentication request **MUST** carry a CSRF token with it, either by a request header or a request query parameter. The CSRF protection design in Documentum REST Services conforms to [Synchronizer Token Pattern](#).

```
// client request with CSRF protect token in the request headers
// this sample assumes 'DOCUMENTUM-CSRF-TOKEN' is set as CSRF token
header name
GET /dctm-rest/repositories/REPO HTTP/1.1
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.3.1 (java 1.5)
Cookie: DOCUMENTUM-CLIENT-TOKEN="ACa/0u...W58=" ← client token
DOCUMENTUM-CSRF-TOKEN: {csrfTokenValue} CSRF token

//server response
HTTP/1.1 200 OK
Content-Type: application/vnd.emc.documentum+json

.....
```

```
//client request with CSRF protect token in query parameters
// this sample assumes 'csrf-token' is set as CSRF token query
parameter name
GET /dctm-rest/repositories/REPO?csrf-token= {urlEncodedCsrfTokenValue} HTTP/1.1
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.3.1 (java 1.5)
Cookie: DOCUMENTUM-CLIENT-TOKEN="ACa/0u...W58=" ← client token
//server response
HTTP/1.1 200 OK
Content-Type: application/vnd.emc.documentum+json

.....
```

## Configuring CSRF token generation

A CSRF token is negotiated between the client and the server. Documentum REST 7.3 supports to generate the CSRF token either from the client side or from the server side. There is a server runtime property to control this.

```
rest.security.csrf.generation.method= server | client
```

When the CSRF token is generated from the **server side**, the response in the initial authentication (Basic, CAS, SAML2, etc.) returns the CSRF parameters and token in response headers. The client is expected to use these parameters for further Client Token authentications.

```
//client sign on
GET /dctm-rest/repositories/REPO.xml HTTP/1.1
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.3.1 (java 1.5)
Authorization: Basic ZG1hZG1pbjpwYXNzd29yZA==

//server response
HTTP/1.1 200 OK
Content-Type: application/vnd.emc.documentum+json
Set-Cookie: DOCUMENTUM-CLIENT-TOKEN="ACa/0u...W58="; ...; HttpOnly
DOCUMENTUM-CSRF-HEADER-NAME: {csrfHeaderName}
DOCUMENTUM-CSRF-QUERY-NAME: {csrfQueryName}
DOCUMENTUM-CSRF-TOKEN: {csrfTokenValue}
```

**basic-ct authentication scheme**

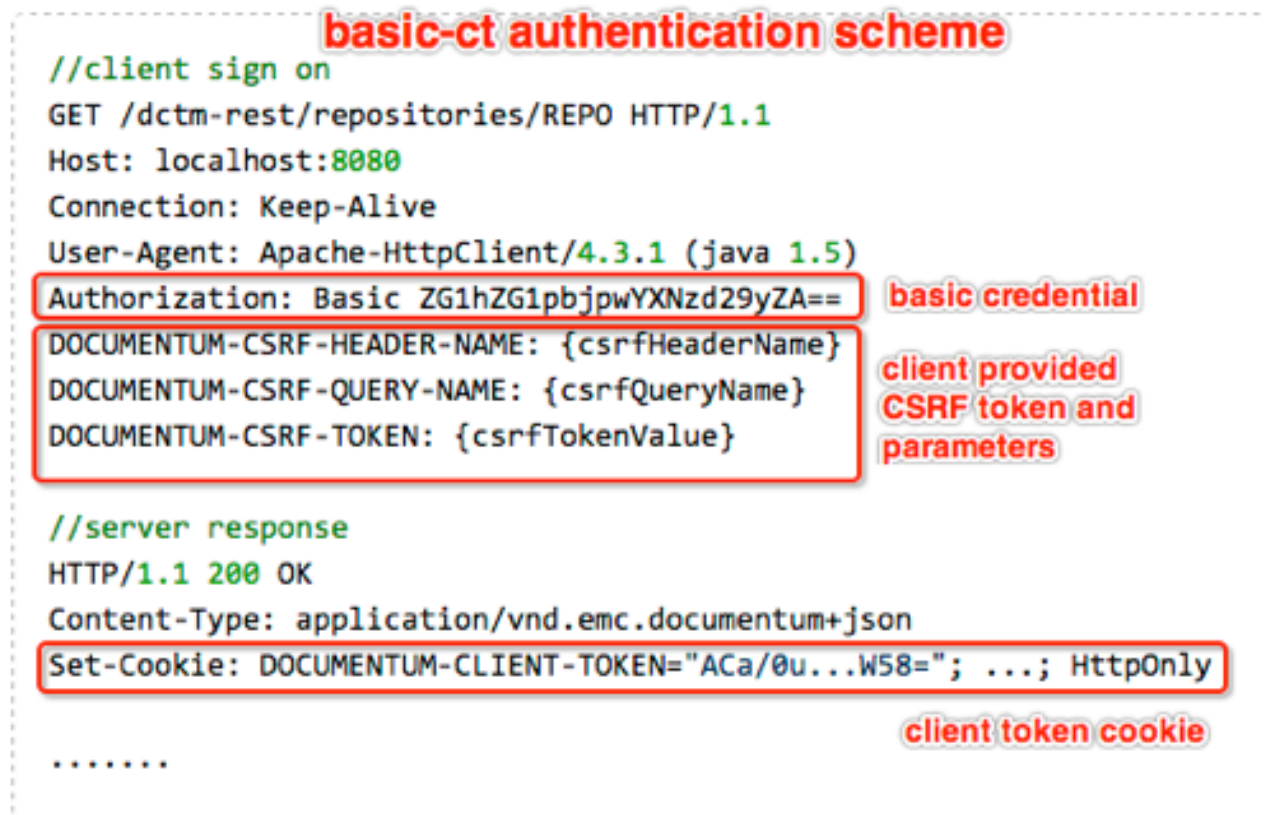
**basic credential**

**client token cookie**

**server generated CSRF token and parameters**

.....

When the CSRF token is generated from the **client side**, the request in the initial authentication (Basic, CAS, SAML2, etc.) provides the CSRF parameters and token in the request header. Documentum REST server takes these parameters into the generation of a Client Token cookie. The client is expected to use these parameters for further Client Token authentication.



## Configuring HTTP method projection

As CSRF attack usually harms for requests that change the server data, a secure HTTP method (GET, OPTIONS, etc.) can be excluded from requiring the CSTF token during Client Token authentications. Documentum REST 7.3 provides another server runtime property to control which HTTP methods are required for CSRF token.

rest.security.csrf.http\_methods= POST,PUT,DELETE,PATCH

## Cross-origin resource sharing (CORS)

Normally JavaScript clients should make requests in the same origin for a website. This prevents JavaScript from making requests across domain boundaries, and has spawned various hacks for making cross-domain requests. [Cross-origin resource sharing \(CORS\)](#) introduces a standard mechanism that can be used by all browsers for implementing cross-domain requests. Here is a simple scenario.

1. Alice visits a website on **domain-a.acme.com**
2. The server returns a main HTML web page that contains resource access to the other domain **domain-b.amce.com**

```
<button onclick="callOtherDomain()">ClickMe</button> <script> var invocation = new XMLHttpRequest(); var url = 'http://domain-
```

3. Alice gets errors on clicking the button **ClickMe**

The failure is normal because CORS prevents cross domain access from **domain-a.acme.com** to **domain-b.acme.com**. Documentum REST 7.3 provides fine-grained CORS configurations to control REST request access from other domains. By default, cross domain access is denied.

## Making cross-origin requests

Most modern web browsers have already supported CORS on the client side. CORS uses HTTP headers to control access to the remote resource. On the client side, the web browser adds a request header **Origin: <http://other-domain.com>** with the request. The server returns a CORS-specific header with the response, **Access-Control-Allow-Origin: <http://some-domains.com>**, which denotes the

origin domains allowed to make requests to the API. A “\*” would mean all domains are allowed.

Here is an example.

```
GET http://localhost:8080/dctm-rest/repositories HTTP/1.1 Accept: */* Authorization: Basic ZG1hZG1pbjpwYXNzd29yZA== Origin: h
```

Many JavaScript clients send CORS **preflight** requests with HTTP method **OPTIONS** before performing the actual requests for the safety. The mechanism is similar except for an additional preflight request ahead.

```
OPTIONS http://localhost:8080/dctm-rest/repositories HTTP/1.1 Accept: application/json Origin: http://other-domain.com Access-Co
```

## Configuring CORS on server side

As mentioned earlier, cross domain access is denied by default. To enable it, Documentum REST server provides runtime properties to allow CORS access for specific domains or all domains. The value is false by default, meaning CORS is disabled. For the detail usage of these parameters, please refer to *Documentum REST Services Development Guide*.

```
rest.cors.enabled= false | true rest.cors.allowed.origins= rest.cors.allowed.methods= rest.cors.allowed.headers= rest.cors.allow.cred
```

## Request sanitization

**Stored cross-site scripting** allows for the permanent injection of JavaScript code. This is a security vulnerability because this JavaScript code can result in the theft of user sessions.

Request sanitization in Documentum REST Services 7.3 cleans up user input to secure against this vulnerability. Request sanitization looks at two parts of the input:

- The input object metadata
- The input HTML content

Here is an example of scripting injection in object metadata. When the stored property value for **object\_name** is rendered into a HTML page, the unexpected alert box will be pop up.

```
{ "name" : "document", "type" : "dm_document", "properties" : { "object_name" : "The<script>alert('I shouldn't be here!')</script>" }
```

When creating documents or other objects in Documentum REST Services, the server can sanitize both the metadata and the HTML content and remove any suspicious scripting code from the text. Here is the result of the object metadata sanitization for the above JSON.

```
{ "name" : "document", "type" : "dm_document", "properties" : { "object_name" : "TheDocument", } }
```

For HTML content sanitization, by default only **html** and **pub\_html** formats are sanitized. But you can still add other formats into content sanitization by server runtime configuration.

## Configuring sanitization on server side

Sanitization is turned off by default because of its performance impact. To turn it on, Documentum REST server provides runtime properties to configure the sanitization for metadata and content, respectively. For the detail usage of these parameters, please refer to *Documentum REST Services Development Guide*.

```
rest.sanitize.type.metadata= false | true rest.sanitize.type.content= false | true rest.sanitize.content.max.size= 500000 rest.sanitize.c
```



Please note Documentum REST Services uses [OWASP AntiSamy](#) library to sanitize the text. The sanitization policies can be configured as well by creating your own policy files. *Documentum REST Services Development Guide* has all the details.

## Sanitizing custom resources

In a custom REST service development with [Documentum REST SDK](#), you may want to enable the sanitization for your resource models. Documentum REST SDK provides a Java class **com.emc.documentum.rest.binding.SanitizeConfig** which allows to customize sanitization on your own resource models. Here are examples.

```
// enforce sanitization @SerializableType(...) public class MyModel { @SerializableField @SanitizeConfig(enforceSanitize=true)
```

## HTTP strict transport security (HSTS)

Some website require HTTPS/SSL protocol on access, but allow HTTP to HTTPS redirect for client requests. This may result into man-in-the-middle attack vulnerability. Here is an example.

1. Alice visits a website [http:// mybank.acme.com](http://mybank.acme.com) with her credentials in **clear** text
2. Server sends a redirect (302) response to [https:// mybank.acme.com](https://mybank.acme.com)
3. Alice visits to [https:// mybank.acme.com](https://mybank.acme.com) with her credentials in **encrypted** text

As you observe, if a man is hijacking Alice's socket packages in the middle, he can easily obtain Alice's credentials in clear text. In this case, although mybank.acme.com has strengthened its communication protocol to HTTPS, it doesn't protect users from leaking confidential information to attackers.

[HTTP Strict Transport Security \(HSTS\)](#) is a web security policy mechanism which helps to protect websites against protocol downgrade attacks and cookie hijacking. If a server requires HTTPS connections, it sends a header **Strict-Transport-Security: <time\_span>** in its response, telling the client that all further requests during that time span period should be issued in HTTPS. When a client supports HSTS, it remembers this setting in it with an expiration policy. Since the timeout could be long (e.g. one year), web browsers usually save the setting in offline store.

Documentum REST Services 7.3 adds server configuration parameters allowing to set HSTS response headers when the protocol is in HTTPS. The value is false by default, meaning HSTS is disabled. For the detail usage of these parameters, please refer to *Documentum REST Services Development Guide*.

```
rest.security.headers.hsts.disabled = false | true rest.security.headers.hsts.include_sub_domains = rest.security.headers.hsts.max_age
```

When HSTS is turned on, the HTTPS response from REST server will include the **Strict-Transport-Security** header.

```
HTTP/1.1 200 OK Content-Type: application/vnd.emc.documentum+json Strict-Transport-Security: max-age=31536000 ; includeSubdomains
```

## Cacheable HTTPS response

Unless directed otherwise, browsers may store a local cached copy of content received from web servers. It is more likely that messages transferred over HTTPS/SSL contain sensitive information. [Cache poisoning](#) vulnerability may occur in such a case. To prevent sensitive

information caching on client side, a response in HTTPS can indicate to stop client caching by headers **Cache-Control**, **Pragma** and **Expires**.

In release 7.3, Documentum REST Services adds server runtime configuration parameter to disable cache headers. The value is false by default, meaning cache prevention over HTTPS is enabled.

```
rest.security.headers.cache_control.disabled = false | true
```

When caching is disabled for HTTPS response, the REST server response is like this.

```
HTTP/1.1 200 OK Content-Type: application/vnd.emc.documentum+json Cache-Control: no-cache, no-store, max-age=0, must-revalidate
```

## Browser cross-site scripting (XSS)

**Cross-site scripting (XSS)** attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user.

1. Alice often visits the search website [http:// acme.com/find?q=](http://acme.com/find?q=) for web search
2. If there are no result for the search item, the website returns an error "404 Not Found for URL /find?q=<term>"
3. One day Alice receives a mail which provides a hidden link to the search website with URL [http:// acme.com/find?q=<script type='text/javascript'>alert\('xss'\);</script>](http://acme.com/find?q=<script type='text/javascript'>alert('xss');</script>).
4. Alice clicks on the link and gets a pop up message box "alert('xss')".

5. If this is a malicious script written by an attacker, Alice's confidential information may be stolen silently.

One approach to prevent XSS attack is to prevent web browsers from running the scripts on the client when receiving a potentially malicious response. [XSS Filter](#) is one approach for that. When a dedicated response header "**X-XSS-Protection**" is set, the client will normalize the received response message, or even stop rendering the page.

Documentum REST Services 7.3 adds server configuration parameters to enable XSS protection. The value is false by default, meaning XSS protection is enabled. For the detail usage of these parameters, please refer to *Documentum REST Services Development Guide*.

```
rest.security.headers.xss_protection.disabled = false | true rest.security.headers.xss_protection.explicit_enable = rest.security.head
```

When XSS protection is enabled (false), the REST response message adds the HTTP header **X-XSS-Protection**.

```
HTTP/1.1 200 OK Content-Type: application/vnd.emc.documentum+json X-XSS-Protection: 1; mode=block ...
```

## Cross-frame scripting (XFS)

[Cross-frame scripting \(XFS\)](#) is an attack that combines malicious JavaScript with an iframe that loads a legitimate page in an effort to steal data from an unsuspecting user. Here is a simple example.

1. Alice often visits her bank web site [http:// mybank.acme.com](http://mybank.acme.com)

2. One attacker creates an evil site and sends a mail to Alice to check her account with the hidden link [http:// mybank.evill.com](http://mybank.evill.com), the link points to a page hides an **iframe** like below.

```
<iframe style="position:absolute;top:-9999px" src="http://mybank.acme.com.com/flawed-page.html    ?q=<script>document.write('<
```

3. Alice clicks on the link and she see everything the same as mybank.acme.com

4. Alice logins and checks her bank account

5. Alice's confidential information is stolen

One approach to prevent frame scripting vulnerability is to prevent web browsers rendering pages within a **<frame>**, **<iframe>** or **<object>**. Mozilla proposed the response header **X-Frame-Options** to do that.

In release 7.3, Documentum REST Services adds server runtime configuration parameter to disable frame scripting. The value is false by default, meaning frame scripting prevention is enabled.

```
rest.security.headers.x_frame_options.disabled = false | true rest.security.headers.x_frame_options.policy =
```

When frame scripting prevention is enabled (disabled as false), the REST response message adds the header **X-Frame-Options**.

```
HTTP/1.1 200 OK Content-Type: application/vnd.emc.documentum+json X-Frame-Options: DENY ...
```

## Response content sniffing

Content sniffing is the practice of inspecting the content of a byte stream to attempt to deduce the file format of the data within it. It is useful when there is not a suitable mechanism between the client and server to detect the content MIME. However, it opens up a serious security vulnerability, in which, by confusing the MIME sniffing algorithm, the browser can be manipulated into interpreting data in a way that allows an attacker to carry

out operations that are not expected by either the site operator or user, such as cross-site scripting.

In Documentum REST Services, the client and server negotiates the content MIME by **Accept** and **Content-Type** headers, so in most cases, there is no need for content sniffing. In release 7.3, Documentum REST Services provides a server runtime configuration parameter to stop content sniffing. It's false by default, meaning content sniffing prevention is enabled.

```
rest.security.headers.content_type_options.disabled = false | true
```

When content sniffing is prevented (true), the response from REST server will include the **X-Content-Type-Options** header.

```
HTTP/1.1 200 OK Content-Type: application/vnd.emc.documentum+json X-Content-Type-Options: nosniff ...
```

## Relevant posts

- [Configure HTTPS/SSL for Documentum REST Services](#)

[Learn more about Documentum REST Services >>](#)