

WHITE PAPER

WEB CACHE DECEPTION ATTACK

Omer Gil

July 2017

Table of Contents

ABSTRACT.....	1
INTRODUCTION.....	1
About caching	1
Servers' reactions.....	2
WEB CACHE DECEPTION METHODOLOGY.....	3
IMPLICATIONS.....	3
CONDITIONS.....	4
KNOWN WEB FRAMEWORKS	4
PHP	4
Django	5
ASP.NET	7
KNOWN CACHING MECHANISMS.....	9
Cloudflare.....	9
IIS ARR	10
NGINX.....	11
MITIGATIONS	13
SUMMARY.....	14
ACKNOWLEDGMENTS.....	14
REFERENCES	14

ABSTRACT

Web cache deception is a new web attack vector that affects various technologies, such as web frameworks and caching mechanisms. Attackers can use this method to expose private and sensitive information of application users, and in certain cases may be able to leverage this attack to perform a complete account takeover.

The attack takes advantage of default behaviors and poor configurations of various technologies that are involved in the application's architecture.

A user who accesses an innocent-looking URL to the domain of a vulnerable application causes the accessed page, with the user's private information, to be stored by a caching mechanism that serves the web application.

INTRODUCTION

About caching

Websites often use web cache functionality in order to reduce web server latency and to provide users with their requested content faster. Instead of letting the web server deal with each request over and over, the caching mechanism stores those application files that are frequently retrieved.

Files that are commonly cached are static, public files: style sheets (css), scripts (js), text files (txt), images (png, bmp, gif), and so on. These files don't usually contain any sensitive information. As can be found in various best practices articles about web cache configuration, it's recommended to cache all static files that are meant to be public, and disregard their HTTP caching headers.

There are several ways to implement caching. For example, caching is performed on browsers: the file is cached, and for a certain period of time, the browser won't ask the web server for the cached file again. This type of caching is NOT relevant for the web cache deception attack.

Another way to implement caching, that is relevant for this attack, is over a server that stands between the client and the web server and functions as a caching mechanism. This type of mechanism can have various forms:

- **CDN (Content Delivery Network).** A distributed network of proxies whose purpose is to serve content quickly. The client will be served from a group of proxies, preferably the closest one to him.
- **Load balancer.** In addition to their job of balancing the traffic between more than one server, load balancers can also cache content in order to reduce the servers' latency.
- **Reverse proxy.** A proxy server that retrieves resources from the web server on behalf of the client, and can cache some of the application's content.

Having reviewed the different forms of caching mechanisms, let's see a demonstration of how web caching actually works. In the following example, the <http://www.example.com> website is served by a reverse proxy. Like any website, this website uses images, CSS files, and scripts to be used publicly. These are all static files used by all or many users in the website, and they return exactly the same content for all users. They do not contain any user information, and are therefore not considered sensitive in any way.

The first time a static file is accessed, the request goes through the proxy. The caching mechanism is not familiar with this file, so it asks the web server for it, and the web server returns the file. Now, the caching mechanism needs to identify the type of the received file. Each caching mechanism works in a different way, but in many cases, the server fetches the file's extension from the end of the URL, and then, according to that mechanism's caching rules, decides whether or not to cache the file.

If the file is cached, the next time any client asks for that file, the caching mechanism already has it stored, so it sends it to the client without asking the web server for it.

Servers' reactions

The web cache deception attack counts on similar browsers' and web servers' reactions, in the same way as the RPO attack, explained in two blogs: The Spanner¹ and XSS Jigsaw².

What happens when a user accesses a URL like <http://www.example.com/home.php/nonexistent.css>, where [home.php](#) is an actual page, and [nonexistent.css](#) doesn't exist?

In this case, the browser sends a GET request to that URL. The interesting thing to look at is how the web server interprets this request. Depending on the server's technology and configuration, the web server might return a 200 OK response with the content of the [home.php](#) page, meaning the URL stays the same.

The HTTP response headers will match the [home.php](#) page: same caching headers and same content type (text/html, in this case).

Response from <http://www.example.com:80/account.php/nonexistent.css> (127.0.0.1)

Forward Drop Intercept is on Action

Raw Headers Hex HTML Render

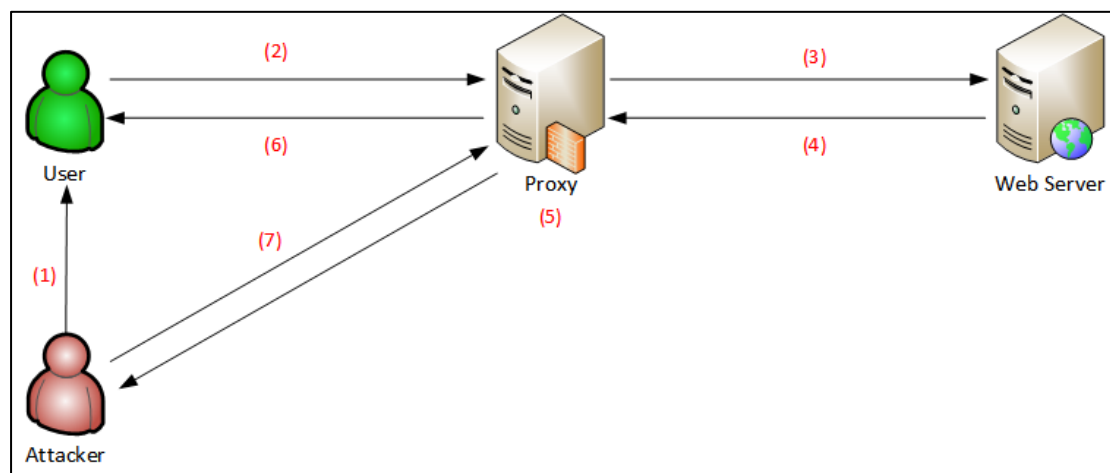
```
HTTP/1.1 200 OK
Date: Thu, 15 Jun 2017 18:47:53 GMT
Server: Apache/2.4.4 (Win32) OpenSSL/0.9.8y PHP/5.4.16
X-Powered-By: PHP/5.4.16
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Length: 1330
Connection: close
Content-Type: text/html

<html>
  <head>
    <title>Account</title>
```

WEB CACHE DECEPTION METHODOLOGY

An unauthenticated attacker can easily exploit this vulnerability, as shown in the following steps:

1. The attacker lures a logged-on user to access <https://www.bank.com/account.do/logo.png>.
2. The victim's browser requests <https://www.bank.com/account.do/logo.png>.
3. The request arrives to the proxy, which is not familiar with this file, and therefore asks the web server for it.
4. The web server returns the content of the victim's account page with a 200 OK response, meaning the URL stays the same.
5. The caching mechanism receives the file and identifies that the URL ends with a static extension (.png). Because the mechanism is configured to cache all static files and disregard any caching headers, the imposter .png file is cached. A new directory named account.do is created in the cache directory, and the file is cached with the name logo.png.
6. The user receives his account page.
7. The attacker accesses <https://www.bank.com/account.do/logo.png>. The request arrives to the proxy server, which directly returns the victim's cached account page to the attacker's browser.



IMPLICATIONS

A successful exploitation will cause the vulnerable page – containing the user's personal content – to be cached and thus publicly accessible. This is a cached, static version of the file; the attacker cannot impersonate the victim. The file cannot be overridden, and remains valid until it expires.

The impact can increase significantly if the body of the response contains the user's session identifier (for some reason), security answers, CSRF tokens, etc. This can be leveraged to additional attacks, and lead to a complete account takeover.

CONDITIONS

Three conditions must be met in order for an attacker to perform web cache deception:

1. When accessing a page like <http://www.example.com/home.php/nonexistent.css>, the web server returns the content of [home.php](#) for that URL.
2. Web cache functionality is set for the web application to cache files by their extensions, disregarding any caching headers.
3. The victim must be authenticated while accessing the malicious URL.

KNOWN WEB FRAMEWORKS

One condition of the attack relates to the application's interpretation for requests of existing URLs with the addition of name of a nonexistent file at the end, like

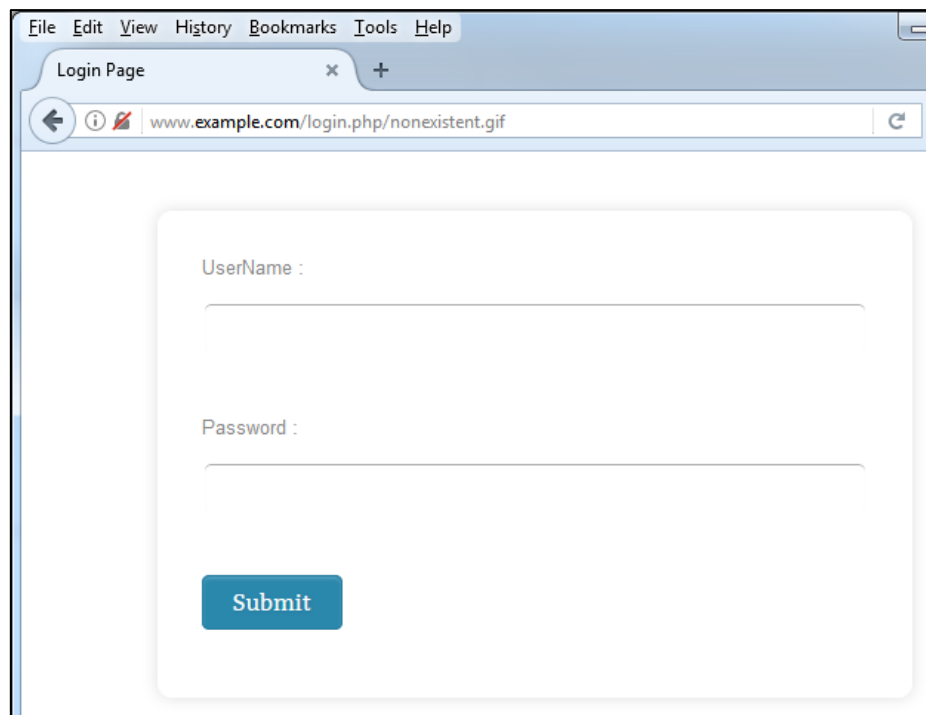
<http://www.example.com/home.php/nonexistent.css>.

The following examples show how the exploit can be performed in several known web frameworks, including an overview of their default behavior, and an explanation of their configuration and workflow.

PHP

By creating a 'pure' PHP application, without using a framework, the application disregards any addition at the end of the URL, and returns a 200 OK response, with the content of the actual page.

For example, when accessing <http://www.example.com/login.php/nonexistent.gif>, the application returns the content of [login.php](#), meaning it meets the first attack condition.



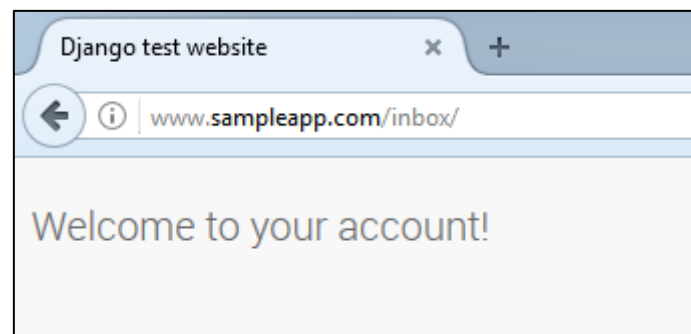
Django

Requests in Django go through a dispatcher which is implemented using the *urls* files. In these files, regular expressions can be set to identify the requested resource in the URI and return the relevant content.

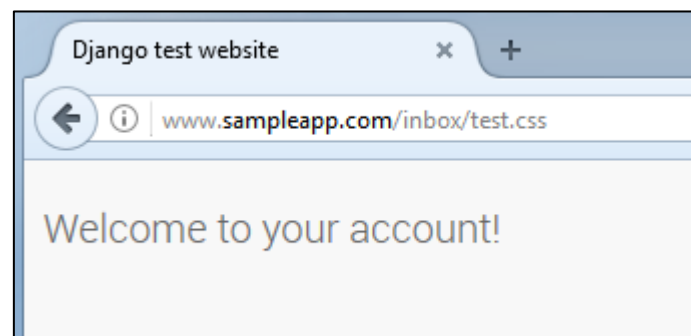
```
from django.conf.urls import include,url
from . import views

urlpatterns = [
    url(r'^inbox/', views.index, name='index')
]
```

This is a common configuration, which returns the content of the Inbox page for a request such as <http://www.sampleapp.com/inbox/>.



This regular expression also matches the same URL with the addition of a nonexistent file like <http://www.sampleapp.com/inbox/test.css>. The result of this regex is that the Django application meets the attack condition.

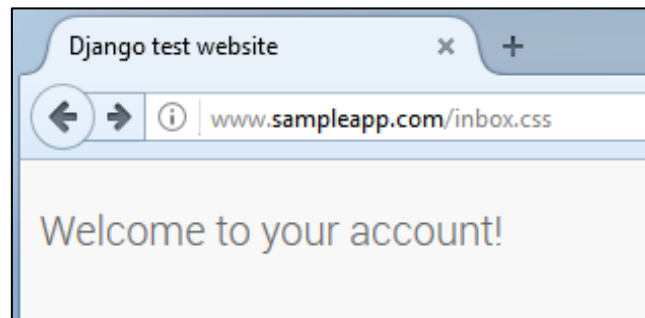


Another example of a vulnerable regex that might be used is by omitting the trailing slash after 'Inbox'.

```
from django.conf.urls import include,url
from . import views

urlpatterns = [
    url(r'^inbox', views.index, name='index')
]
```

With this regex implemented, in addition to finding a match in the standard <http://www.sampleapp.com/inbox> URL, it also finds a match in <http://www.sampleapp.com/inbox.css>.



When the dollar sign is appended to the regex, the application won't find a match in triggering URLs.

```
from django.conf.urls import include,url
from . import views

urlpatterns = [
    url(r'^inbox/$', views.index, name='index')
]
```



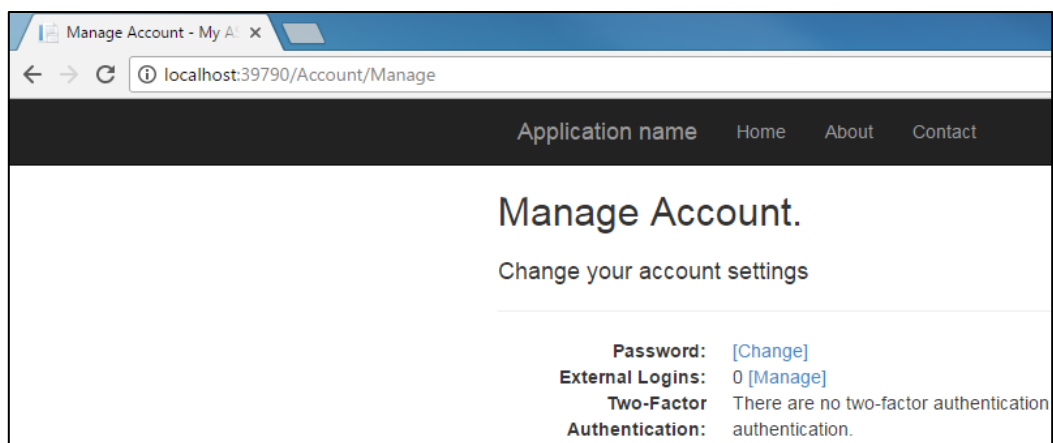
ASP.NET

In the ASP.NET framework there is a built-in feature called *FriendlyURLs*, whose main purpose is to make the URL 'cleaner' and friendlier. For accessing <https://www.example.com/home.aspx>, the application removes the extension and redirects users to <https://www.example.com/home>.

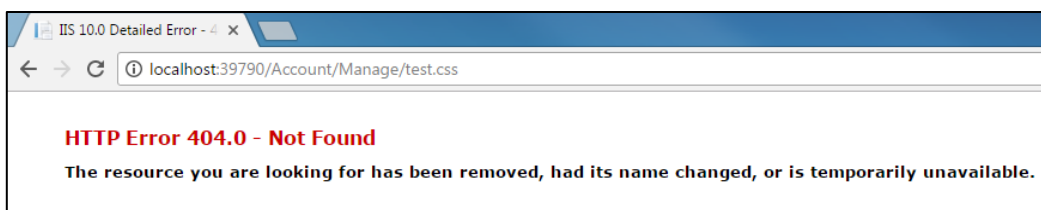
This feature can be configured in the *Route.config* file and it is turned on by default in any ASP.NET application.

```
7 namespace WebApplication7
8 {
9     public static class RouteConfig
10    {
11        public static void RegisterRoutes(RouteCollection routes)
12        {
13            var settings = new FriendlyUrlSettings();
14            settings.AutoRedirectMode = RedirectMode.Permanent;
15            routes.EnableFriendlyUrls(settings);
16        }
17    }
18 }
```

When the *FriendlyURLs* feature is turned on, and a user accesses the existing [Manage.aspx](#) page by requesting <http://localhost:39790/Account/Manage.aspx>, the .aspx extension is removed and the page content is displayed.



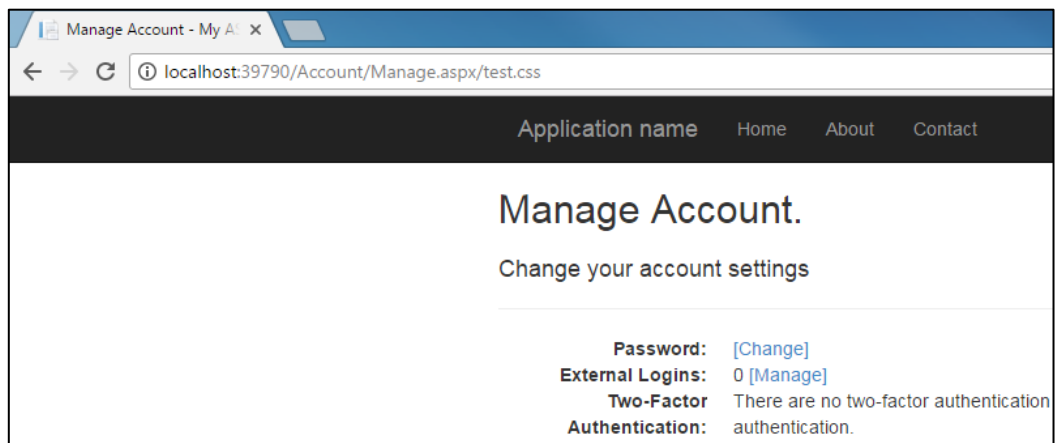
Accessing a triggering URL with this configuration <http://localhost:39790/Account/Manage.aspx/test.css> results in the .aspx extension being omitted; the user is redirected to <http://localhost:39790/Account/Manage/test.css>, which causes a 404 error. This means that when *FriendlyURLs* is turned on, the application does NOT meet the attack condition.



Although *FriendlyURLs* is turned on by default, there are many websites that don't use this feature. It can be easily turned off in *Route.config*.

```
7 namespace WebApplication7
8 {
9     public static class RouteConfig
10    {
11        public static void RegisterRoutes(RouteCollection routes)
12        {
13            var settings = new FriendlyUrlSettings();
14            settings.AutoRedirectMode = RedirectMode.Off;
15            routes.EnableFriendlyUrls(settings);
16        }
17    }
18 }
```

When the feature is turned off, accessing the same triggering URL results with a 200 OK response, and the actual content of the *Manage.aspx* page is returned.



KNOWN CACHING MECHANISMS

The second condition of the attack is for the web cache functionality to be set for the web application to cache files according to the extension at the end of the URL, disregarding any caching headers. Following are examples of several known caching mechanisms, with explanations of their caching process and how they identify the type of a received file.

Cloudflare

When a file arrives to Cloudflare servers from a web server, it goes through two phases. The first phase is called the Eligibility Phase, in which Cloudflare checks whether the caching feature is applied on the website and on the directory the file came from. If the answer is yes (and it probably is, as this is probably why the website used Cloudflare's services from the outset), the Cloudflare server checks whether the URL ends with one of the following static extensions:

```
class, css, jar, js, jpg, jpeg, gif, ico, png, bmp, pict, csv, doc, docx, xls, xlsx, ps, pdf, pls, ppt, pptx, tif, tiff, ttf, otf, webp, woff, woff2, svg, svgz, eot, eps, ejs, swf, torrent, midi, mid
```

If it does, the file moves on to the second phase – the Disqualification Phase, in which the Cloudflare server checks for the existence of HTTP caching headers.

Unfortunately, by accessing a triggering URL, the web server returns the caching headers of the existing dynamic page, meaning that the file will probably return with a no-cache directive.

Fortunately, Cloudflare has a feature called 'Edge cache expire TTL', which provides the option to override any existing headers. By setting this feature to 'on', files that return from the web server with a no-cache directive will be cached. As a result of Cloudflare's recommendation to use this header for a variety of reasons, it is commonly used.

Create a Page Rule for webcachedeception.com

If the URL matches: By using the asterisk (*) character, you can create dynamic patterns that can match many URLs, rather than just one. [Learn more here](#)

Then the settings are:

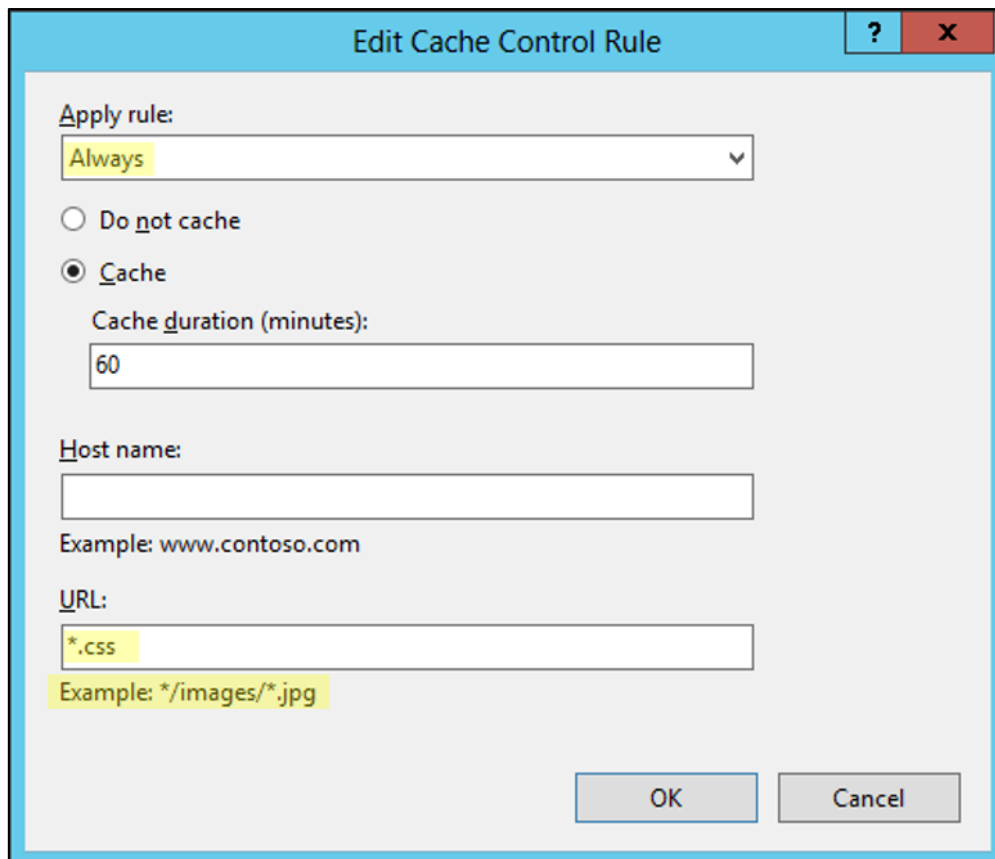
<input type="text" value="Cache Level"/>	<input type="text" value="Standard"/>	<input type="button" value="X"/>
<input type="text" value="Edge Cache TTL"/>	<input type="text" value="2 hours"/>	<input type="button" value="X"/>

IIS ARR

ARR (Application Request Routing) provides load balancing capabilities to IIS.

One of the features that ARR offers is caching. Cache rules can be set for web servers supported by the load balancer, in order to save files to the cache directory. When creating a new cache rule, we define the file type to be cached using a wildcard and the desired extension. When a file arrives to ARR, it looks for this pattern in the file's URL. Effectively, ARR identifies the file type according to the extension at the end of the URL.

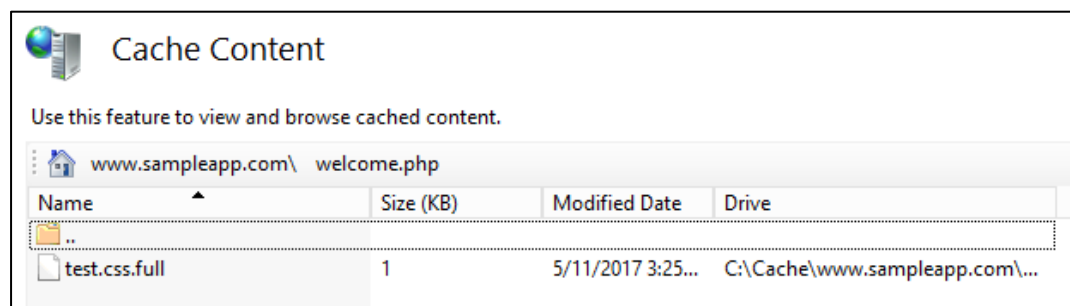
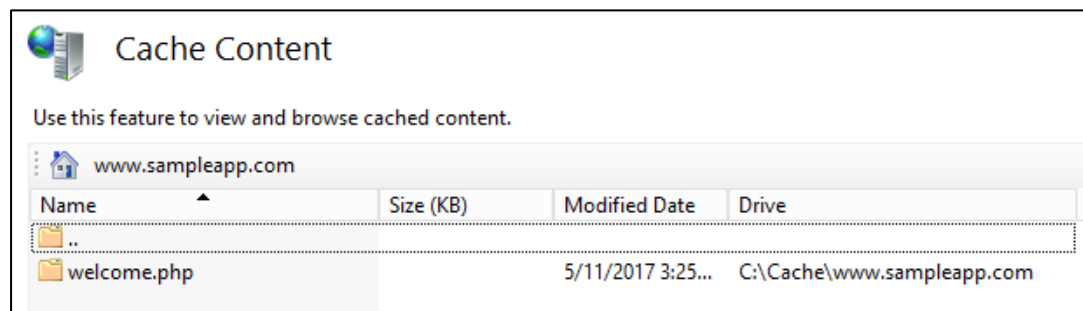
In addition, IIS ARR contains an option to disregard the file's caching headers, causing the rule to be applied in any case.



In the following example, IIS ARR is linked with two web servers and is configured to cache all stylesheets and JavaScript files.

Cache Control Rules				
Use this feature to configure rules to manage cache control directives.				
Cache	Host Name	URL	Condition	Cache Duration
Yes		*.css	Always	60
Yes		*.js	Always	10080

A request to a triggering URL (<http://www.sampleapp.com/welcome.php/test.css>) results in the creation of a new directory named [welcome.php](#) in the cache directory, and inside it, a new file named [test.css](#), which contains the content of the user's [welcome.php](#) page.



NGINX

An NGINX server that functions as a load balancer provides caching capabilities to store pages that return from the web servers.

The caching rules can be configured in NGINX configuration files. The following example shows a configuration that instructs NGINX to cache specific types of static files, and to disregard their caching headers.

```
location ~* \.(css|js|gif|png)$ {
    proxy_cache my_cache;
    proxy_cache_valid 200 60m;
    proxy_pass http://[redacted];
    proxy_ignore_headers Expires Cache-Control Set-Cookie;
}
```

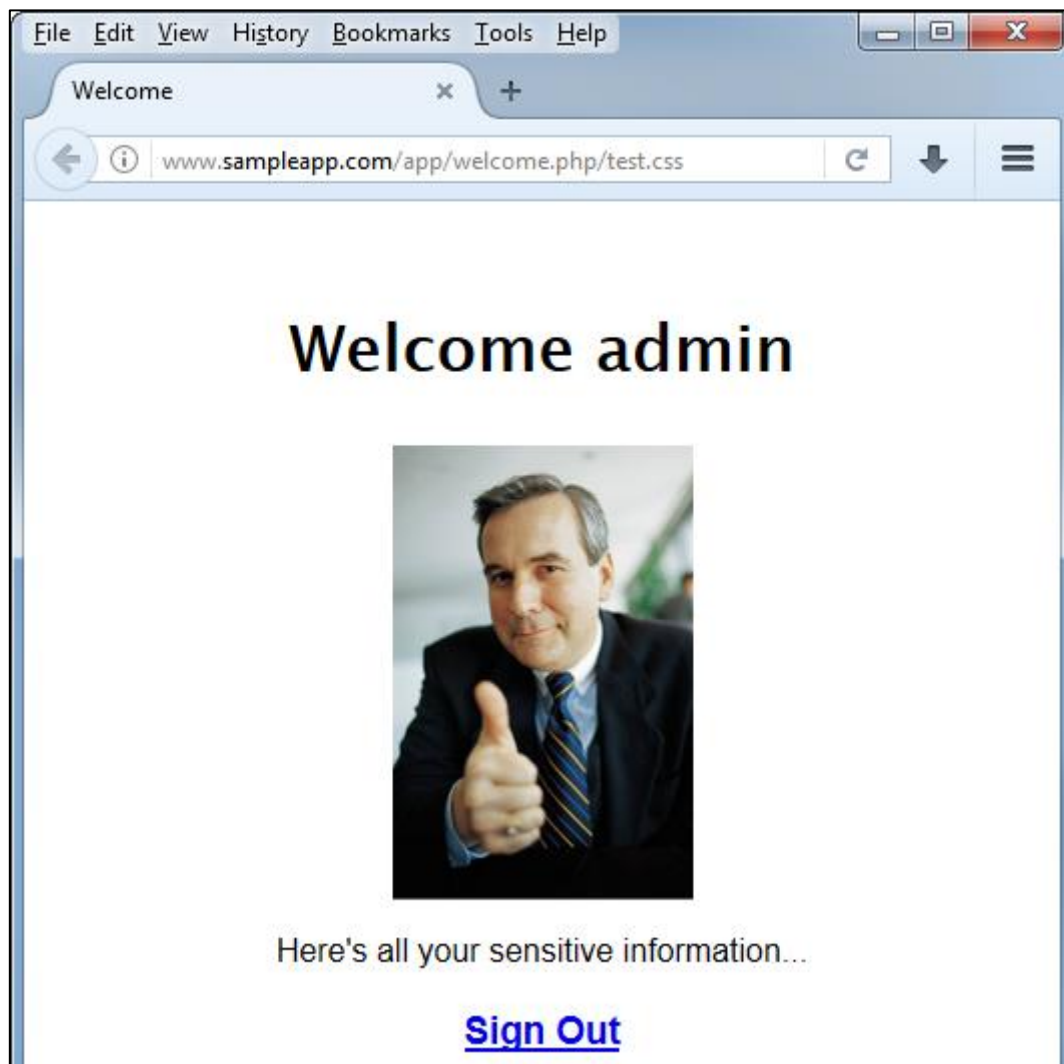
When a page arrives to NGINX from the web server, NGINX searches for the extension at the end of the URL, and identifies the file type according to it.

At first, nothing is cached in the cache directory.

```
root@[redacted]:/cache# tree
.
└── temp

1 directory, 0 files
```

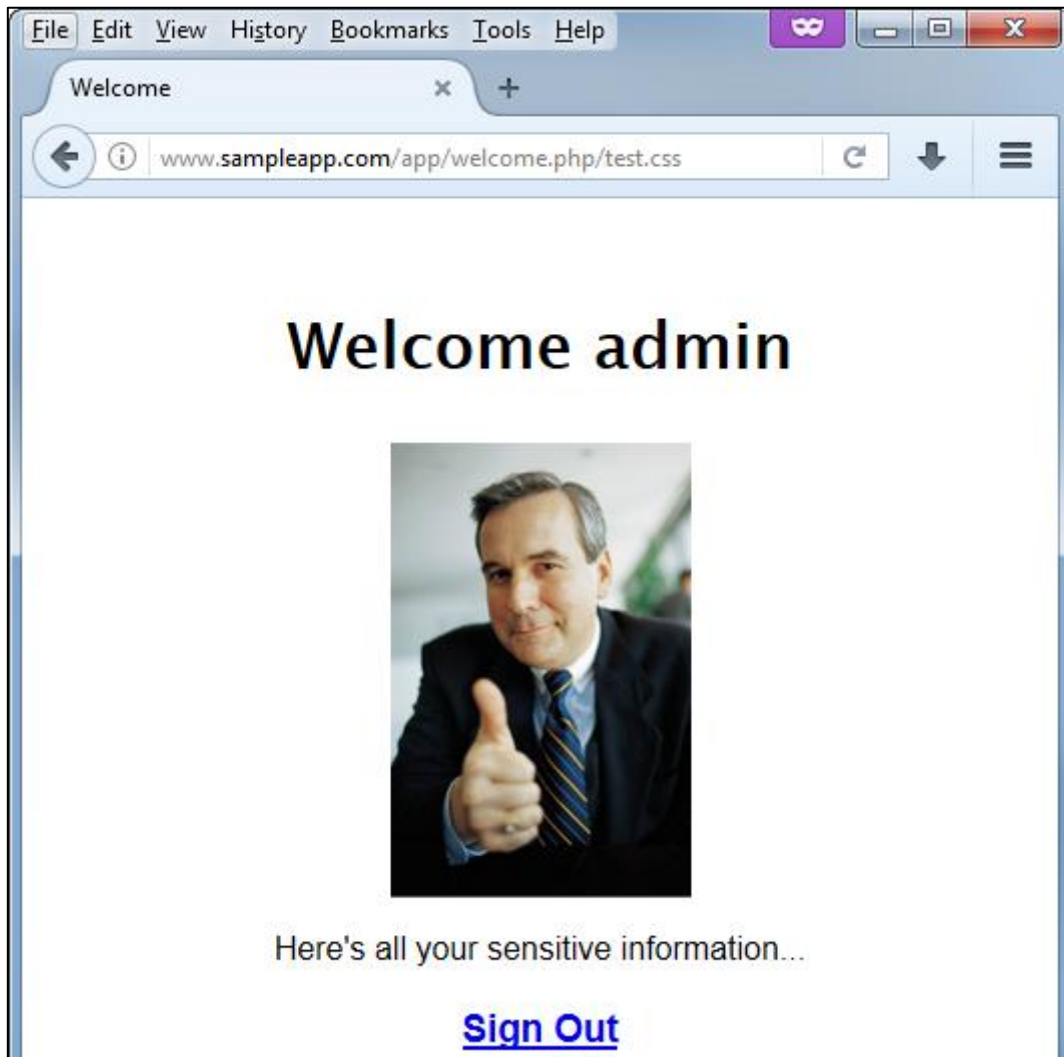
An authenticated user accesses a triggering URL (<http://www.sampleapp.com/app/welcome.php/test.css>), and the user's page is cached in the cache directory.



```
root@ [REDACTED] :/cache# tree
├── f
│   ├── de
│   │   └── 8d30ee601f9befd24e22993bae090def
│   └── temp
│       ├── 1
│       │   └── 00
└── [REDACTED]
```

5 directories, 1 file

Then, an attacker accesses the triggering URL while unauthenticated, and the NGINX server returns the cached file, containing the user's private content.



MITIGATIONS

1. Configure the cache mechanism to cache files only if their HTTP caching headers allow.
2. Store all static files in a designated directory and cache only that directory.
3. If the cache component provides the option, configure it to cache files by their content type.
4. Configure the web server so that for pages such as <http://www.example.com/home.php/nonexistent.css>, the web server does not return the content of [home.php](#) with the triggering URL; instead, the server should return a 404 or 302 response.

SUMMARY

Web cache deception is an attack that is not only easy to perform, but can also have serious consequences, from exposing users' personal information, to attackers gaining control over users' accounts. A number of well-known websites were found to be vulnerable to this attack; most of these websites were served by the most common CDNs. It's safe to assume that there are still many sites that could fall victim to the attack.

Although this White Paper relates only to a limited sample of technologies that can meet the web cache deception attack conditions, there are various additional web frameworks and caching mechanisms that could provide attackers with similar opportunities to perform the attack.

The web frameworks and caching mechanisms that create the conditions for this vulnerability are not in and of themselves vulnerable; the main issue is one of improper configurations.

To prevent web cache deception attacks, technical personnel should first and foremost be aware of the conditions that can enable this attack. Furthermore, vendors would be advised to make efforts to prevent their products from meeting these conditions. This can be achieved by disabling features, changing default settings and behaviors, and providing warnings to increase the awareness of technical personnel.

ACKNOWLEDGMENTS

Sagi Cohen, Bill Ben Haim, Sophie Lewin, Or Kliger, Gil Biton, Yakir Mordehay, Hagar Livne

REFERENCES

1. RPO – The Spanner blog
<http://www.thespanner.co.uk/2014/03/21/rpo/>
2. RPO gadgets – XSS Jigsaw blog
<http://blog.innerht.ml/rpo-gadgets/>
3. Django URL dispatcher
<https://docs.djangoproject.com/en/1.11/topics/http/urls/>
4. NGINX caching
<https://serversforhackers.com/c/nginx-caching>
5. Web cache deception attack – original blog
<http://omergil.blogspot.co.il/2017/02/web-cache-deception-attack.html>
6. Web cache deception attack in PayPal home page
<https://www.youtube.com/watch?v=pLte7SomUB8>
7. Understanding our cache and the web cache deception attack – Cloudflare blog
<https://blog.cloudflare.com/understanding-our-cache-and-the-web-cache-deception-attack/>
8. On web cache deception attacks – Akamai blog
<https://blogs.akamai.com/2017/03/on-web-cache-deception-attacks.html>