# RainLoop Webmail - Emails at Risk due to Code Flaw

BY SIMON SCANNELL | APRIL 19, 2022

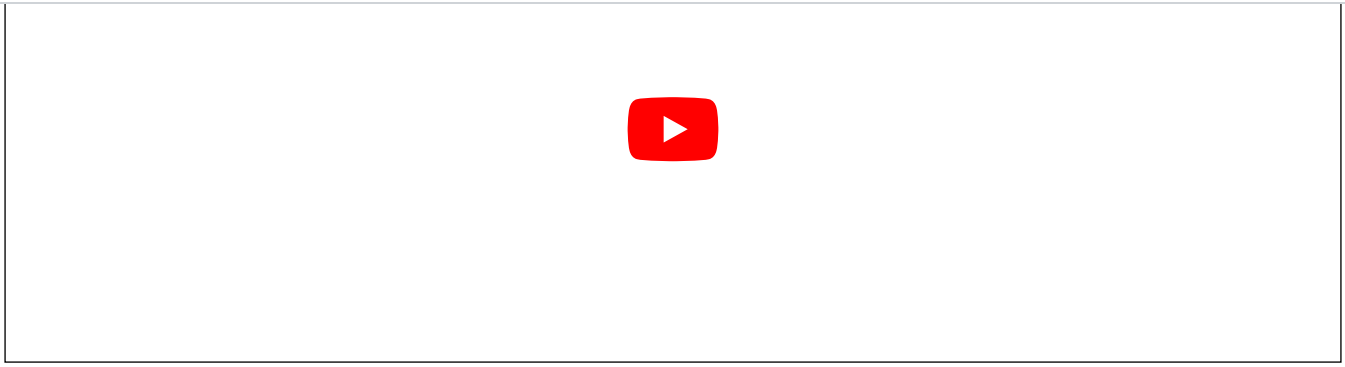
Security

RainLoop is an open-source webmail client used by thousands of organizations to exchange sensitive messages and files via email. In this blog post, we are warning RainLoop users about a code vulnerability that allows attackers to steal emails from the inboxes of victims. At the time of writing, no official patch is available.

The code vulnerability described in this blog post can be easily exploited by an attacker by sending a malicious email to a victim that uses RainLoop as a mail client. When the email is viewed by the victim, the attacker gains full control over the session of the victim and can steal any of their emails, including those that contain highly sensitive information such as passwords, documents, and password reset links. Let's have a look what happened and what we can learn from it.

## Impact

The discovered code flaw is a Stored Cross-Site-Scripting vulnerability (CVE-2022-29360) that affects the latest version v1.16.0 of RainLoop. At the time of writing, no official patch is available. The vulnerability can be exploited in any RainLoop installation that runs with default configurations. An attacker who knows the email address of an employee of a targeted organization can send the victim a maliciously crafted email. When it is viewed in the webmail interface, it executes a hidden JavaScript payload in the browser of the victim. No further user interaction is required.



## Technical Details

In the following sections, we go into detail about the Stored Cross-Site-Scripting vulnerability and how gadgets were abused to make JavaScript run automatically once a victim views a malicious email.

### Stored XSS in the email body (CVE-2022-29360)

RainLoop's backend is a PHP application that acts as a proxy between a user and their mail server. Similar to mail clients, such as Thunderbird, it enables a user to log into a mail server, fetch emails, view them, and send emails.

# Sanitization Logic

As RainLoop is a web application, it needs to render incoming emails to HTML code. It also needs to ensure that the rendered HTML code has been validated and does not contain malicious components (e.g. unsafe links, JavaScript tags).

On a high level, RainLoop deploys the following flow to achieve this:

1. Receive the raw, untrusted HTML code from the mail server
2. Create an instance of the built-in `DOMDocument` class in PHP. This parses HTML into a tree structure of HTML elements and their attributes
3. Depending on the configuration, use an allow or deny list to remove any dangerous contents in the tree structure
4. Convert the sanitized tree structure of the `DOMDocument` into HTML code

Intuitively it makes sense to analyze the code that attempts to remove any dangerous HTML code (step 3 in the above's list) and find a weakness inside of that code to bypass the sanitizer. However, our experience has shown there are often logic bugs **after** the sanitization steps have been performed. From the security researcher's point of view, they are much easier to spot and are often overlooked by developers: for good examples of previous findings using this pattern, see Zimbra Stored XSS and WordPress CSRF to RCE.

We mentioned that the 4th step converts the tree structure of the `DOMDocument` into HTML code. Usually, this step is trivial as the `DOMDocument` class has the built-in `saveHTML()` method which does exactly what is required.

## Faking a HTML <body>



ered to the user: due to normalization
ins `<html>` and `<body>` tags. Although this is

Additionally, `<body>` tags might contain important attributes such as styles and classes that must be preserved. RainLoop solves these problems by parsing the attributes from the `<body>` tag of the email structure and then wrapping the HTML code of the email in a fake body that contains the original `<body>` attributes.

In the following paragraphs, we will describe how this process works in RainLoop, show the corresponding code snippets and finally describe a logic flaw in this process that leads to a Stored XSS vulnerability.

In the first step, RainLoop fetches references to the `<html>` and `<body>` nodes from the tree structure and then calls `saveHTML()` on all children to get the sanitized HTML code without `<html>` and `<body>` tags:

**rainloop/v/0.0.0/app/libraries/MailSo/Base/HtmlUtils.php**

```
222    $oHtml = $oDom->getElementsByTagName('html')->item(0);
223    $oBody = $oDom->getElementsByTagName('body')->item(0);
224
225    foreach ($oBody->childNodes as $oChild)
226    {
227        $sResult .= $oDom->saveHTML($oChild);
228    }
```

In the next step, the attributes of the `<html>` node are fetched and added to a newly created `<div>` tag to simulate the `<html>` tag:

**rainloop/v/0.0.0/app/libraries/MailSo/Base/HtmlUtils.php**

```
232    $aHtmlAttrs = HtmlUtils::GetElementAttributesAsArray($oHtml);
233    $aBodylAttrs = HtmlUtils::GetElementAttributesAsArray($oBody);
234
235    $oWrapHtml = $oDom->createElement('div');
236    $oWrapHtml->setAttribute('data-x-div-type', 'html');
237    foreach ($aHtmlAttrs as $sKey => $sValue)
238    {
239        $oWrapHtml->setAttribute($sKey, $sValue);
240    }
```

This process is repeated for the `<body>` tag, but with an important difference: The `<div>` tag that is created to preserve the `<body>` attributes is created with the text content `___xxx___`. This fake `<body>` is then appended to the fake `<html>` node and dumped to HTML code:

**rainloop/v/0.0.0/app/libraries/MailSo/Base/HtmlUtils.php**

```
242    $oWrapDom = $oDom->createElement('div', '___xxx___');
243    $oWrapDom->setAttribute('data-x-div-type', 'body');
244    foreach ($aBodylAttrs as $sKey => $sValue)
245    {
246        $oWrapDom->setAttribute($sKey, $sValue);
247    }
248
```



Let's walk through this code with an example. Let's assume an attacker sent the following email:

```
    <p>Wehope you are doing good!</p>
</body>
</html>
```

The process we described thus far would then yield the following HTML code, stored in the `$sWrp` variable:

```
<div data-x-div-type="html">
    <div data-x-div-type="body" data-some-attr="abc">
        ___xxx___
    </div>
</div>
```

In the final step, the rest of the email is inserted in the wrapping code above. This is done by replacing the `___xxx___` inside of the fake wrapping body with the previously generated HTML code:

**rainloop/v/0.0.0/app/libraries/MailSo/Base/HtmlUtils.php**

```
252     $sResult = \str_replace('___xxx___', $sResult, $sWrp);
```

This would finally yield the following HTML code:

```
<div data-x-div-type="html">
    <div data-x-div-type="body" data-some-attr="abc">
        <h1>Hello!</h1>
        <p>I hope you are doing good!</p>
    </div>
</div>
```

# The Logic Bug

As an attacker can control the attributes of a `<body>` tag and their values, they could create a `<body>` tag with an attribute value of `___xxx___`.

This could, for example, result in the following HTML markup:

```
<div data-x-div-type="html">
    <div data-x-div-type="body" data-some-attr="___xxx___">
        ___xxx___
    </div>
</div>
```

As `str_replace()` replaces the `___xxx___` string as many times as it can find, an attacker can insert controlled user input into the quoted value of the `data-some-attr`. Let's assume an attacker crafted an email as follows:

```
<body data-some-attr="___xxx___">
<div title="x onclick='alert(document.cookie);//' y">
    XSS PoC
</div>
```



xxx___ with the rest of the HTML code:

## Patch

At the time of writing, no official patch is available. We recommend the RainLoop fork SnappyMail. It has great security improvements and is actively maintained. We would like to thank the maintainers of this fork for their quick response and analysis of this issue. They confirmed to us that they are not affected. For this reason, we recommend users of RainLoop migrate to SnappyMail in the long term.

To help in the short term, we encourage users to apply the following inofficial patch that we developed (please carefully use at your own risk):

```
--- /tmp/HtmlUtils.php  2022-04-11 09:34:35.000000000 +0200
+++ rainloop/v/0.0.0/app/libraries/MailSo/Base/HtmlUtils.php    2022-04-11 09:35:12.000000000 +0200
@@ -239,7 +239,8 @@
                $oWrapHtml->setAttribute($sKey, $sValue);
            }
-           $oWrapDom = $oDom->createElement('div', '___xxx___');
+           $rand_str = base64_encode(random_bytes(32));
+           $oWrapDom = $oDom->createElement('div', $rand_str);
            $oWrapDom->setAttribute('data-x-div-type', 'body');
            foreach ($aBodylAttrs as $sKey => $sValue)
            {
@@ -250,7 +251,7 @@
            $sWrp = $oDom->saveHTML($oWrapHtml);
-           $sResult = \str_replace('___xxx___', $sResult, $sWrp);
+           $sResult = \str_replace($rand_str, $sResult, $sWrp);
        }
       $sResult = \str_replace(\MailSo\Base\HtmlUtils::$KOS, ':', $sResult);
```

In order to use this patch:

1. Create a backup of your RainLoop files!
2. Upload the patch file contents above to a file called `rainloop_xss.patch` and store it in the root directory of your RainLoop installation
3. Run the following command:

```
patch rainloop/v/1.13.0/app/libraries/MailSo/Base/HtmlUtils.php < rainloop_xss.patch
```

**Please note that your path may vary, depending on the version of RainLoop you use. In the example above, version 1.13.0 is used. Make sure to use the correct version in your path.**

## Timeline

| Date | Action |
| --- | --- |
| 2021-11-30 | We request a security contact by contacting support@rainloop.net. No response |

sponse

m them of our 90-day disclosure policy. No re

In this blog post, we analyzed a Persistent Cross-Site-Scripting vulnerability in RainLoop that triggers when a victim views a maliciously crafted email. The vulnerability occurred due to a logic bug **after** the sanitization process, which is often overlooked by security audits. We have found similar bugs in high-profile targets such as Zimbra and WordPress. In general, we recommend developers to not modifying any data after it has been sanitized, as any modification could reverse the sanitization step. Additionally, it is recommended to work with a DOM tree object, rather than operating on HTML text, as this leaves much more room for mistakes.

## Related Blog Posts

- WordPress CSRF to RCE
- MyBB From Stored XSS to RCE
- Zimbra Webmail compromise via email
- SmartStore.net Malicious message leading to eCommerce takeover

SIMON SCANNELL
Vulnerability Researcher