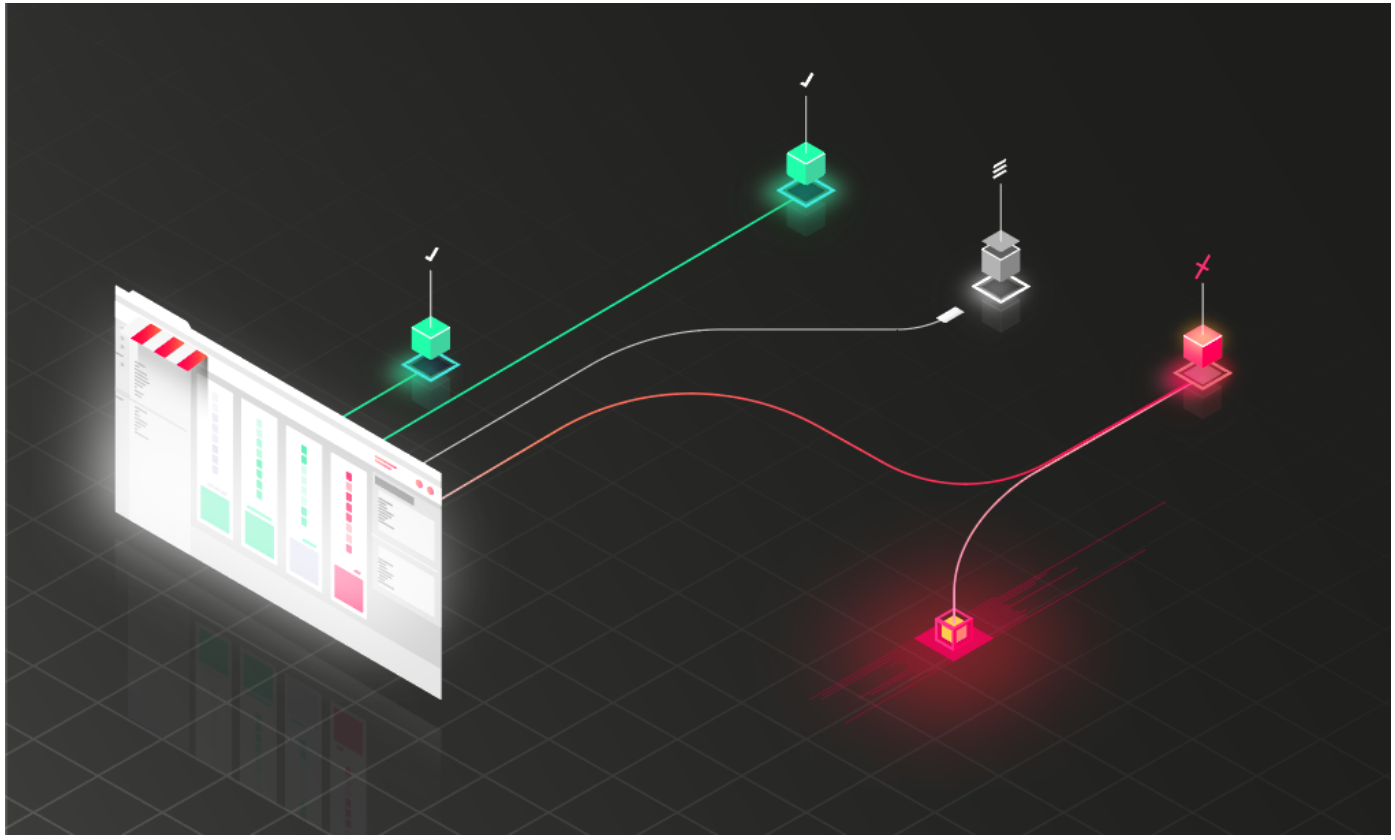
Magento 2.3.1: Unauthenticated Stored XSS to RCE

11 min read — 2 Jul 2019 by Simon Scannell

This blog post shows how the combination of a HTML sanitizer bug and a Phar Deserialization in the popular eCommerce solution Magento **<=2.3.1** lead to a high severe exploit chain. This chain can be abused by an unauthenticated attacker to fully takeover certain Magento stores and to redirect payments.

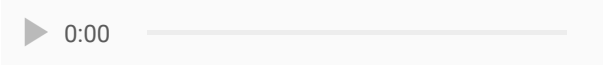


PHP MAGENTO REMOTE CODE EXECUTION CROSS SITE SCRIPTING PHAR DESERIALIZATION VULNERABILITY ECOMMERCE

Impact

A successful attack enables an unauthenticated adversary to persistently inject a JavaScript payload into the administrator backend of a Magento store. When triggered, this JavaScript

payload can then perform automated exploit steps in the browser of a victim. We visualize these steps in our video in form of our JavaScript-based *RIPS shell*.



When an employee of the store logs into the admin dashboard, the injected JavaScript payload runs and hijacks the administrative session of the employee. An authenticated Remote Code Execution vulnerability is then exploited, which results in a full takeover of the store by the attacker. The attacker could then cause financial harm to the company running the store. For example, the attacker could redirect all payments to his bank account or steal credit card information.

The vulnerabilities can be exploited if a Magento store uses the built-in, core Authorize.Net payment module, a *Visa* solution that allows for the processing of credit card payments. Please note that Authorize.Net is not responsible for any of the vulnerabilities, but rather the implementation in Magento. Since Authorize.Net is a popular payment processing service for credit cards, the exploit chain affects many Magento stores. Considering that testing if a target store uses the Authorize.Net module is easy and can be automated, mass exploitation is possible.

We rate the severity of the exploit chain as **high**, as an attacker can exploit it without any prior knowledge or access to a Magento store and no social engineering is required. Considering that all Magento stores transact over **$155 billion** annually, attackers are highly motivated to take advantage of such vulnerabilities.

Who is affected?

Affected are all Magento stores that have the Authorize.Net module enabled and run a vulnerable version, which are listed in the following table:

Branch	Patched in version	Vulnerable versions
2.3	2.3.2	<= 2.3.1
2.2	2.2.9	<= 2.2.8
2.1	2.1.18	<= 2.1.17

Technical analysis

In the following we analyze two distinct security vulnerabilities that can be chained. Due to the severity of these issues, certain exploit details are omitted on purpose.

Unauthenticated Stored XSS

Magento provides multiple sanitization methods for different purposes. This section is going to detail a bypass for the `escapeHtmlWithLinks()` sanitization method and how a bypass

lead to an unauthenticated Stored XSS vulnerability in the cancellation note of a new product order.

However, before discussing said method, it makes sense to first get some background knowledge on Magento sanitization and understand it's main sanitization method, `escapeHTML()`:

vendor/magento/framework/Escaper.php

```
50  /**
51   * Escape string for HTML context.
52   *
53   * AllowedTags will not be escaped, except the following: script, img, embe
54   * iframe, video, source, object, audio
55   *
56   * @param string|array $data
57   * @param array|null $allowedTags
58   * @return string|array
59   */
60  public function escapeHtml($data, $allowedTags = null)
```

All you have to know about `escapeHTML()` is that it parses user input (`$data`) and removes all HTML tags that are not specified with the second parameter, `$allowedTags`, from the user input string. If the second parameter is not set, the entire user input string will simply be escaped. The method furthermore allows only a few HTML attributes to bet set in each allowed tag, namely `id`, `class`, `href`, `style` and a few others.

We did not come up with a bypass for `escapeHTML()`, so we searched for code that acts on user input after it was sanitized with `escapeHTML()`, as modification of sanitized data often

leads to vulnerabilities. We found the method `escapeHtmlWithLinks()`. The next paragraphs and snippets will explain how this method works and how a logic flaw in it lead to a XSS vulnerability.

The purpose of `escapeHtmlWithLinks()` is to remove all HTML tags except for a whitelisted set of tags from a user input string. The difference to `escapeHTML()` is that it will additionally remove all attributes except the `href` attribute from `<a>` tags within the user input string, in order to make links extra secure.

As the following code snippet shows, `escapeHtmlWithLinks()` starts off by parsing all `<a>` tags within the user input string into an array (`$matches`):

vendor/magento/module-sales/Helper/Admin.php

```
150  public function escapeHtmlWithLinks($data, $allowedTags = null)
151  {
152      ⋮
153      $data = str_replace('%', '%%', $data);
154      $regexp = "#(?J)<a"
155      ."(?:(?:\s+(?:(?:href\s*=\s*(['\"])(?<link>.*?)\\1\s*)|(?:\S+\s*=\s*(
156      ."\>?(?:(?:(?<text>.*?)(?:<\/a\s*>?|(?=<\w))|(?<text>.*)))#si";
157      while (preg_match($regexp, $data, $matches)) {
158          ⋮
```

The next step is to sanitize the text of the link and the URL contained in the `href` attribute. This is done by recreating a minimalistic tag (line 164 - 169 in the next code snippet).

The resulting sanitized link is then stored in the `$links` array, which will be used later. `escapeHtmlWithLinks()` then replaces the original `<a>` tag, which was just sanitized, with a

`%$is` within the user input string, where `$i` is simply the number of the replaced `<a>` tag.

```
156         :
157         while (preg_match($regexp, $data, $matches)) {
158         $text = '';
159         if (!empty($matches['text'])) {
160             $text = str_replace('%%', '%', $matches['text']);
161         }
162         $url = $this->filterUrl($matches['link'] ?? '');
163         //Recreate a minimalistic secure a tag
164         $links[] = sprintf(
165             '<a href="%s">%s</a>',
166             htmlspecialchars($url, ENT_QUOTES, 'UTF-8', false),
167             $this->escaper->escapeHtml($text)
168         );
169         $data = str_replace($matches[0], '%' . $i . '$s', $data);
170         ++$i;
171         }
```

To give a concrete example of what is described above and shown in the last code snippet, here is what would happen to an example user input at this stage of the sanitization method:

`<i>Hello, <a href="/the-world/" title="Hello World">World!</a></i>` would turn into `<i>Hello, %1s</i>`

After `escapeHtmlWithLinks()` has replaced all `<a>` tags with a corresponding `%s` in the user input string, it will pass the resulting user input to `escapeHTML()`. This will sanitize the user input securely (line 172 of the next snippet). However, it will then insert the sanitized links back into the now sanitized string via `vsprintf()`. This is where the XSS vulnerability occurs. We will discuss how exactly the XSS vulnerability works in the following paragraph.

```
170         ⋮
171         } // End of while
172         $data = $this->escaper->escapeHtml($data, $allowedTags);
173         return vsprintf($data, $links);
```

The issue with simply inserting the sanitized links into the escaped user input string is that `escapeHtmlWithLinks()` does not care about the position of an `<a>` tag within a string. The following table demonstrates how this can lead to a *HTML attribute injection*.

Step	User input string

Parse <a> tags from user input string	`<i id=" <a href='http://onmouseover=alert(/XSS/)'>a link</a> ">`
	`a malicious link`
	`</i>`

Replace <a> tags with a %s	`<i id=" %1s ">`
	`a malicious link`
	`</i>`

Step	

Remove all unwanted tags from user input string	`<i id=" %1s ">`
	`    a malicious link`
	`</i>`

Insert sanitized <a> tags into sanitized string	`<i id=" <a href="http://onmouseover=alert(/XSS/)>">a link</a> ">`
	`    a malicious link`
	`</i>`

As can be seen in the above table, the `<a>` tag is replaced with a `%1s` and the user input string is then sanitized. As `%1s` is not a dangerous value, it passes the sanitization step. When `escapeHtmlWithLinks()` then reinserts the sanitized link with `vsprintf()`, an additonal double quote is injected into the `<i>` tag, which allows for an attribute injection.

This allows an attacker to inject arbitrary HTML attributes into the resulting string. By injecting a malicious `onmouseover` event handler and a `style` attribute to make the link an invisible overlay over the entire page, the XSS payload triggers as soon as a victim visits a page that contains such an XSS payload and moves his mouse.

The `escapeHtmlWithLinks()` method is used to sanitize order cancellation notes that are created when a user starts the order process with Authorize.Net but then cancels it. By abusing the bypass described above, an attacker can inject arbitrary JavaScript into the order overview of a just cancelled order. When an employee then reviews the cancelled order, the XSS payload triggers.

Authenticated Phar Deserialization

Once an attacker has hijacked the session of an authenticated user, he can abuse a Phar Deserialization vulnerability within the controller that is responsible for rendering images within the WYSIWYG editor. The following code snippet shows how the POST parameter `__directive` is passed to the `open()` method of an image adapter class. This method internally passes the user input to the function `getimagesize()`, which is vulnerable for Phar deserialization (find out more in our post: What is Phar Deserialization).

vendor/magento/module-cms/Controller/Adminhtml/Wysiwyg/Directive.php

```
53  public function execute()
54     {
55          $directive = $this->getRequest()->getParam('___directive');
56          $directive = $this->urlDecoder->decode($directive);
57          ⋮
58          $image = $this->_objectManager->get(\Magento\Framework\Image\Adapte
59          try {
60              $image->open($imagePath);
61          ⋮
```

By injecting a `phar://` stream wrapper into an image file handler, an attacker can trigger a PHP object injection. He can then chain POP gadgets from the Magento core that in the end lead to Remote Code Execution.

Timeline

Date	What
2018/09/25	We report a Stored XSS vulnerability in Magento 2.2.6.
2018/11/28	Magento releases a patch for the Stored XSS vulnerability in 2.2.7 and 2.1.16.
2018/12/13	We report a bypass for the patch in Magento 2.3.0
2019/01/11	We report the Phar Deserialization vulnerability to the Magento security team.
2019/01/26	We discover that the Stored XSS can be triggered by unauthenticated attackers on Magento stores with a certain configuration. We inform Magento.
2019/01/29	Magento verifies the vulnerability.

Date	What
2019/03/26	Magento releases a security update and fixes the Phar Deserialization in Magento 2.3.1, 2.2.8 and 2.1.17. The Stored XSS vulnerability is not mentioned in the changelogs and no patch is available.
2019/04/09	Magento closes the ticket for the Stored XSS as "Resolved".
2019/04/09	We ask Magento if this issue has been fixed, since no mention of it is in the changelogs and no modifications have been made to the escapeHTMLWithLinks() method.
2019/04/10	Magento reopens the ticket.
2019/06/25	A patch is made available in version 2.3.2, 2.2.9 and 2.1.18

Summary

This blog post detailed how an unauthenticated Stored XSS vulnerability can be combined with an authenticated Phar Deserialization vulnerability to hijack Magento stores on a mass exploitable scale. The technical sections demonstrated that the exploitation of today's security flaws often depends on multiple sanitization, logic and configuration flaws. We highly recommend all users to update to the latest Magento version.

Related Posts

- MyBB <= 1.8.20: From Stored XSS to RCE
- WordPress 5.1 CSRF to Remote Code Execution
- osClass 3.6.1: Remote Code Execution via Image File
- WordPress 5.0.0 Remote Code Execution
- WordPress Design Flaw Leads to WooCommerce RCE

Simon Scannell

Security Researcher

Simon is a self taught security researcher at RIPS Technologies and is passionate about web application security and coming up with new ways to find and exploit vulnerabilities. He currently focuses on the analysis of popular content management systems and their security architecture.