Automatic Examination-Based Whitelist Generation for XSS Attack Detection

doi: 10.1007/978-3-030-02613-4_29

Automatic Examination-based Whitelist Generation for XSS Attack Detection

Keisuke Inoue[1], Toshiki Honda[1], Kohei Mukaiyama[1],
Tetsushi Ohki[1], and Masakatsu Nishigaki[1]

[1] Shizuoka University, Hamamatsu, Japan
nisigaki@inf.shizuoka.ac.jp

Abstract. When faced with cross-site scripting (XSS) attacks, it is difficult to counter all malicious inputs such that they are rendered completely harmless. In such situations, the introduction of a whitelist-based XSS countermeasure is considered to be an effective and robust approach. However, as the behavior of current web applications is complex, it is difficult to theoretically generate the necessary and sufficient whitelists. To this end, we propose an examination-based approach for whitelist generation instead of a theory-based one. We focus on software tests that are always performed during the final stage of the development process and establish a method to automatically generate whitelists that are consistent with the specifications of each web application. By adding the function for whitelist generation on a web application's test tool, a whitelist can be generated without changing the development process of a conventional web application. We implement our proposed method and evaluate its effectiveness.

1 Introduction

Along with improvements in computer performances and broadening of network bandwidths, web contents are shifting from conventional static webpages to dynamic web contents. Currently, the majority of websites use JavaScript libraries such as jQuery [1] and content management system (CMS) tools such as WordPress [2] and are generated dynamically using programs on both the client side and the server side. As a result, a variety of services offered by conventional applications (non-web applications) have become available in dynamic web contents. Simultaneously, vulnerabilities that occur in conventional applications are also present in web services, leading to a rapid increase in cyber attacks that target web applications.

Cross-site scripting (XSS) attacks are cyber attacks that target web applications. XSS is the vulnerability of a web application caused either by a function that checks whether an input value is appropriate or a function that renders harmful input values harmless. An attacker could exploit this vulnerability to cause arbitrary scripts to execute on a user's browser. As a result of such an attack, an attacker steals the session ID and can pretend to be an authorized user and use the website illegally. This also gives the attacker the opportunity to perform a drive-by download attack on the user. Other vulnerabilities of web applications, such as SQL injections, affect the service provider's web servers, whereas an XSS attack directly affects users' PCs. It is, thus paramount that effective additional countermeasures are introduced.

The main cause of this vulnerability is the incompleteness of XSS countermeasures that are enacted when developing web applications. Detoxification by sanitizing is currently used as a general XSS countermeasure [3]. However, the fact that

vulnerabilities have been observed even in the web services of a major IT company that has invested sufficient resources in security countermeasures [4], shows the difficulty of perfectly rendering all malicious inputs completely harmless, regardless of the capabilities of the developer. In addition, new unknown (zero-day) vulnerabilities and attack methods are sometimes discovered, in which case all the existing countermeasures to prevent harm through sanitizing remain insufficient.

For such varied types of attacks, it is effective to adopt a whitelist-based countermeasure and to only allow predefined functions. However, a theory for creating the necessary and sufficient whitelist has not yet been developed, and this has limited the impact of whitelist-based XSS attack countermeasures. A policy-based countermeasure is a typical example. In literature [6][7][8], policy-based methods of countering XSS attacks by restricting script operations has been proposed. However, developers need to determine policies for defining the behavior of scripts, and methodologies to decide necessary and sufficient policies have not yet been developed.

To address this problem, the present paper proposes a shift in strategy such that whitelists are created based on examination rather than theory. To create an examination-based whitelist, we focus on a test process that is carried out as the final stage of web application development. More specifically, our proposed method detects and prevents XSS attacks by defining the behavior confirmed in the test process as a whitelist, and by only allowing the execution of the scripts included in the whitelist at the time of usage of the web application. This method has several advantages which are as follows: (i) the execution of a script is allowed only for the patterns confirmed beforehand in its test process, (ii) a whitelist created through the test process is consistent with the specification since a software test is conducted based on the specification of each web application, and (iii) the whitelist can be automatically generated through the test process. We implement our function to generate whitelists using Selenium [5], which is widely adopted as an automated test tool for web applications, and evaluate the effectiveness of our proposal.

2 XSS Attacks and Mechanisms

2.1 Types of XSS Vulnerabilities

The types of XSS vulnerabilities can be divided into three categories which are outlines in the following subsections. These vulnerabilities are grouped depending on whether they are persistent or non-persistent, and they are processed on the server-side or the client-side.

2.1.1 Reflected XSS

Reflected XSS is a vulnerability that may occur in a web application, where the server-side program receives input from the user as a parameter, generates a webpage according to the parameter, and returns the webpage to the client (Fig. 1). This occurs when the server-side program uses a harmful input value without sanitizing. This is not a persistent attack because the server does not hold its value.

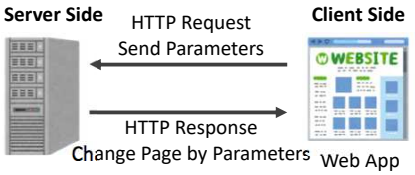

Fig. 1. Overview of the reflected XSS

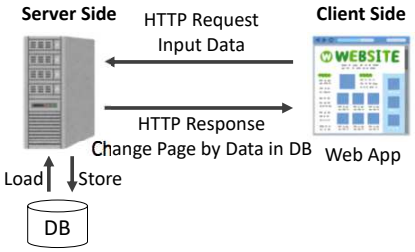

Fig. 2. Overview of the stored XSS

2.1.2 Stored XSS

Stored XSS is a vulnerability that may occur in a web application, where the server-side program receives an input from a user as a parameter, saves it in a storage such as a database (DB), generates a webpage according to the stored parameter, and returns the webpage to the client (Fig. 2). This occurs when the server-side program uses a harmful input value without sanitizing. Once a harmful input value is received by the server as a parameter and stored in the DB of the server, the application reads this saved value every time the application is executed, and thus, there is persistence in such attacks.

2.1.3 DOM-based XSS

Document Object Model (DOM)-based XSS is a vulnerability that may occur in a web application, where the client-side program receives input from the user as a parameter and reflects the parameter in the DOM without sanitizing (Fig. 3). As the value of the received parameter is processed on the client and as the server does not store this value, there is no persistence in such attacks.

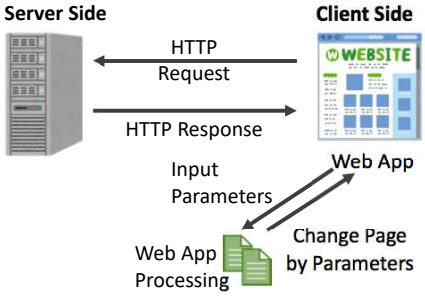

Fig. 3. Overview of DOM-based XSS

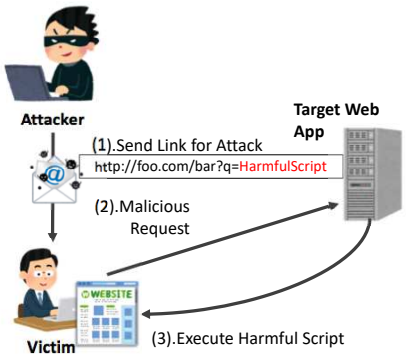

Fig. 4. How Reflected XSS attacks work

2.2 Mechanism of a Reflected XSS Attack

In this section, we describe the mechanism of reflected XSS attacks, which are a typical form of such attacks. These consist of three entities, "attacker," "victim," and a "target web application" with an XSS vulnerability (Fig. 4).

Using a means of communication such as email, the attacker sends the link information (URL of the target web application and parameters exploiting the XSS vulnerability) to the victim to generate a harmful request for the target web application (Fig. 4, (1)). By accessing the link, the victim sends the attacker-designated harmful request to the target web application through the victim's browser (Fig. 4, (2)). The target web application transmits the web content, including the harmful script, as a response to the victim's browser. The victim's browser recognizes the harmful script contained in the response as a DOM element of the web content and executes it (Fig. 4, (3)).

When this vulnerability is exploited, the attacker will be able to execute an arbitrary script on the browser. For instance, by acquiring the session ID of the target web application, the attacker may impersonate the victim on the web application. An attacker could also display a fake login page to steal the victim's information and may perform a drive-by download attack by directing the victim to a malicious website created by the attacker. Such harmful script functions are mostly not displayed on the monitor of the victim's PC. Therefore, victims often do not even notice that they have been attacked.

3 XSS Attacks Countermeasures

3.1 Sanitizing

Harmful scripts of XSS attacks are injected into web applications by malicious parameters embedded in a link by the attacker. Therefore, general XSS countermeasures consist of rendering the input values harmless. Specifically, this comprises sanitization through validation of the input value and escape processing. The sanitization process is programmed by developers when developing a web application.

3.1.1 Validation of Input Value

For an XSS attack to cause a script to execute in the browser, it is necessary to include the code that the browser recognizes as a script in the parameter (e.g., "`<script> harmful script </script>`"). For such parameters, the web application can suppress its execution by checking whether an HTML tag which, indicates a script (e.g., `<script>`), is included. When the parameter includes some script, sanitizing is achieved by invaliding its parameter.

3.1.2 Escape Processing

As an example, if a user wants to intentionally display the character string "`<script> harmful script </script>`" on the browser, and inputs this

string as a parameter of a web application, it is impossible to disable the <script> tags using the input validation method described in the previous section. In this case, it is possible to perform countermeasures such as escape processing, so that the tags are not recognized as DOM elements. If the web application receives a parameter "<script> harmful script </script>", then the web application converts "<" to "<" and ">" to ">". As a result, the response that the web application sends to the browser will be "<script> harmful script </script$gt;", and the browser recognizes this as a character string instead of a script, and displays "<script>harmful script</script>" on the browser.

3.1.3 Problems

In addition to "<script>," browsers have many other tags and events that are recognized as scripts. This means that there are multiple patterns that can avoid these input validations and/or escape processing. Therefore, current sanitizing countermeasures are assumed to be results of frequent imperfections. Considering that similar vulnerabilities have been found in major web services that have invested sufficient resources in security countermeasures, it can be inferred that it is a difficult task to perfectly render all malicious inputs completely harmless, regardless of the efforts made by the developer. In addition, sometimes, new and unknown (zero-day) vulnerabilities and attack methods are discovered. Therefore, all the existing countermeasures to prevent harm through validation and sanitizing are assumed to be insufficient.

3.2 Whitelist

As described in the previous section, the way of describing web contents is diverse and there are multiple patterns that can avoid sanitization. For such situations, one of the most effective and robust strategies is to adopt a whitelist-based countermeasure and to only allow predefined scripts. As a typical instance of whitelist-based countermeasures, policy-based countermeasures, such as BEEP [6], ConScript [7], and CSP [8], have been studied. These methods prevent XSS attacks by prohibiting the execution of scripts against predefined security policy. For whitelist-based countermeasures, it is very important to improve the quality of a whitelist. In this line, as another research direction, there is an empirical approach for enriching a whitelist [9].

3.2.1 Policy-based Whitelists

Jim et al. [6] have proposed browser-enforced embedded policies (BEEP). This is a method of limiting access to the important APIs described in the security policy by rewriting the JavaScript code for the API components on the client's browser. The security policy is described in the application by the developer, and the API calls are hooked and checked on the browser.

Meyerovich et al. [7] have proposed ConScript, which is very similar to BEEP [6]. This is a method of applying a whitelist-type security policy to applications by hooking the API on the client's browser using aspect oriented programming for security-critical API components.

Sid et al. [8] have proposed content security policy (CSP). This is a method of restricting the operation of applications in the browser by adding a security policy to the HTTP response header of the server. CSP is now commercially used and is

available on several browsers and servers. However, it does not seem to be proliferated widely. One of the reasons for this limited use is that CSP does not work as expected on every browser or server due to differences in implementation.

3.2.2 Experience-based Whitelists

Although it is not strictly for XSS attack countermeasures, Kakuta et al. [9] have conducted research on a method of automatically expanding a whitelist and a blacklist for malware infection detection. This is a method of categorizing a website into whitelist, blacklist, or graylist by analyzing the access log of the network and calculating the degree of malignancy of the website. For a website classified as a graylist, additional authentication tests – which are impossible for a malware (automated program) to perform – are applied to assess whether the website is white or black.

3.2.3 Problems

In the policy-based methods, it is necessary for developers to decide on the security policies. This means that developers are required to have reasonable knowledge of web security. Not only that, corpulence of web applications has also become an issue. Since the structure and behavior of the current web applications are very complex, it is not easy for even a developer to accurately grasp the operation of the application. Therefore, policy setting is difficult and prone to error. Such errors can lead to other vulnerabilities and allow an attacker to circumvent the policy. In other words, no theory for creating a necessary and sufficient security policy has yet been developed and hence, policy-based XSS countermeasures have limited effect. The experience-based method can help to enrich whitelists to some extent, but there is still a lack of theoretical foundation in such empirical approaches. This kind of imperfection is a significant problem that all the existing whitelist-based approaches face.

4 Examination-based Whitelist

4.1 Overview

To overcome the problems of existing whitelist-based XSS countermeasures described in the previous section, we propose a shift in the strategy of whitelist generation from theory-based approach to examination-based approach. In this paper, to generate an examination-based whitelist, we focus on the test process that is carried out on the final stage of web application development. The examination-based whitelist guarantees to execute only scripts that were confirmed beforehand in the test process. More specifically, our proposed method detects and prevents XSS attacks by defining the behavior examined in the test process as a whitelist and only allows the execution of the scripts included in the whitelist at the time of usage of the web application. The software test is performed based on the specification document of each web application. Therefore, by integrating the whitelist generation process in the test process, our method manages to automatically generate an examination-based whitelist – that does not deviate from the specification for each script of each web application – without altering the development process of a web application.

This method has the following advantages:

(i) The execution of the script is allowed only for the patterns confirmed beforehand in its test process.
(ii) A whitelist created through the test process is consistent with the specification since a software test is conducted based on the specification of each web application.
(iii) The whitelist can be automatically generated through the test process.

4.2 Corroboration of Examination and Specification

The key concept of the examination-based whitelist is defining only the behavior confirmed beforehand as a whitelist. This is not a theoretically necessary and sufficient whitelist, but we can obtain a heuristically adequate whitelist by corroborating examination and specification.

The primal factor in XSS attacks is the insertion of a script that was not intended by the web application developer. That is, the essence of the problem is that an unintended script is executed. To tackle this, we can use a specification document that is created in the initial stage of software development. A specification document is a document that indicates all the functions implemented and how the software behaves when each function is executed. In other words, a specification is "a set of intended behaviors". The specification document is used in the final stage of web application development, too. Before releasing a web application, a developer verifies whether the web application meets the requested specifications or whether it operates as written in the specification. Thus, it is expected that all the behaviors indicated in the specification are confirmed at the test process, and therefore an examination-based whitelist that includes all the intended behaviors can be generated though the test process.

It should be noted, however, that achieving 100% test coverage is not an easy task. To mitigate this, we focus on the script structure of HTML page source codes. As described before, the primal factor in XSS attacks is a malicious script insertion into HTML page source codes. Therefore, XSS attacks can only be detected by comparing the script structures of the HTML page source codes. Since the script structure of HTML page source codes can only be defined by testing typical input values, it is expected that an exhaustive test is not required to obtain a whitelist. In addition, this contributes to reducing the size of whitelists and simplifies the comparison. That is why, in this paper, we extract only the script parts (JavaScript sections) from HTML page source codes and register them in a whitelist through the test process. An example of a whitelist generated for a specific HTML page source code is shown in Fig. 5.

4.3 Detection Method

In a dynamic web application, the displayed content changes in response to user inputs. In other words, according to the parameters received from a user, the structure of the HTML page source code changes. In the proposed method, the script structures of HTML page source codes that were executed in the test process are all registered as a whitelist. Therefore, the HTML page source codes generated by the web application never deviate from the script structure registered in the whitelist as long as the parameters inputted by a user are within the specification of the web application.

On the other hand, if an attacker inputs parameters that the web application developer did not expect, then an HTML page source code that is not registered in the whitelist will be generated. Typically, an XSS vulnerability exploit parameter will generate an HTML page source code with a new unintended script inserted. As a

result, the script structure of the HTML page source code generated by the web application changes into a structure that is not registered in the whitelist. Our proposal uses this feature to detect XSS attacks.

The XSS attack detection in the proposed method is carried out in the browser of the client side. Whenever a user accesses a web service and inputs a parameter into the web application, the browser always compares the script structure of the HTML page source codes generated by the web application with the script structure of the HTML page source code registered in the whitelist. If the structures do not match, then it can be judged that an XSS attack has occurred.

Fig. 6 shows an example of an XSS attack. By inputting "`<script>alert (“Attack!”)</script>`" into the webpage of Fig. 5, an unintended HTML page source codes of Fig. 6 is generated. As depicted in Fig. 5 and Fig. 6, the structures of the script are different from each other and therefore the attack can be detected.

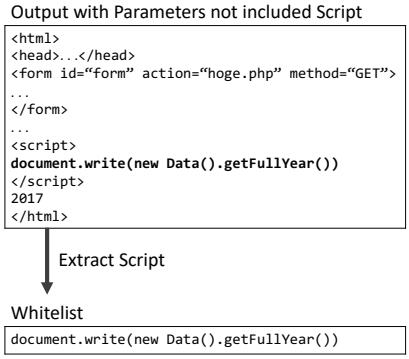

Fig. 5. Generating a whitelist

Fig. 6. Possibility of XSS attack

4.4 Automatic Whitelist Generation

As described in Section 4.2, in this paper we propose to define only the behavior confirmed beforehand as a whitelist, and to use specification documents for obtaining a heuristically adequate examination-based whitelist. The examination-based whitelist is composed of JavaScript sections in the HTML page source codes that the web application generates when the intended parameters indicated in the specification document are inputted. This means that the whitelist generation process of the proposed method is almost identical to the test process of a web application.

The purpose of the test process is to ensure that the web application is implemented according to the specification. When releasing a web application, the developer performs an operation test to verify that the web application is operating according to the specification, including a parameter test. In this parameter test, several patterns of parameters described in the specification document are inputted, and then the correct response of the web application is verified. This means that all the HTML page source

codes required for whitelist generation may be obtained during the test process of the web application.

As we have noted, the proposed method can integrate the whitelist generation process into the test process of a web application. More specifically, the examination-based whitelist of the proposed method can be obtained by extracting all the JavaScript parts from each HTML page source codes generated during the test process. Given this, we propose to implement the automatic whitelist generation function by slightly modifying a web application's test tool. By adopting this function, it becomes possible to generate an examination-based whitelist for each web application through the ordinary web application development process. This can make our whitelist-based XSS attack detection more effective and reasonable. A comparison between the process adopting the proposed function and the ordinary process is shown in Fig. 7.

4.5 Deployment

As explained in the previous section, in our detection method, the structure of the script to be executed is verified on the user-side's client browser. Therefore, to put the proposed method to practical use, the generated whitelist must be sent to the user-side and needs to be available from the client browser. This can be realized by storing the whitelist in the web server along with the corresponding web contents, and sending both the web contents and the whitelist when the browser requests the web service. One of the typical implementations is to embed a hyperlink to the whitelist in the HTML page source code of the corresponding web contents, like the attached CSS files. By doing so, the whitelist is downloaded at the same time when receiving the HTML page source code from the server. The download and verification of the whitelist can be implemented by an extension function for the browser. The method for realizing this is illustrated in Fig. 8.

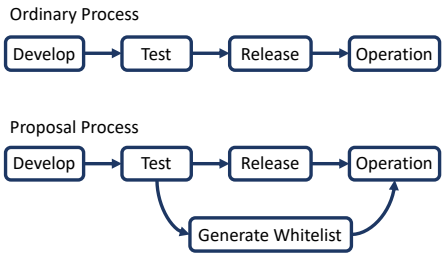

Fig. 7. Comparison of processes

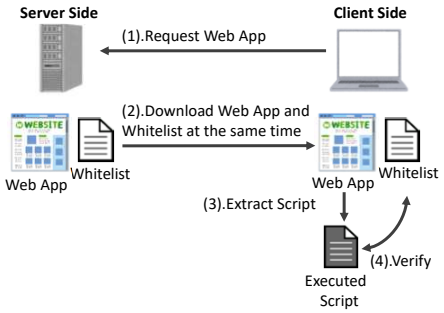

Fig. 8. Deployment of whitelist-based detection

5 Implementation and Evaluation of Proposed Method

We implement an automatic whitelist generation component that is executed on a test tool for web applications, and evaluate the effectiveness of our proposal.

5.1 Target Web Application

As the web application to be tested, we created a webpage imitating an application form (Fig. 9, left). This web application consists of six elements: an application date output by JavaScript, a headline, three text boxes, and a button. By inputting a character string in each text box of the application form and pressing the confirmation button, the page transitions to the application confirmation page (Fig. 9, right), and the inputted value can be checked.

The specification document for this web application is as follows: (Spec. 1) by entering a character string in the text boxes and pressing the confirmation button, the page transition occurs; (Spec. 2) in the transition destination page, "Confirmation" is displayed in the headline and all the inputted items are presented.

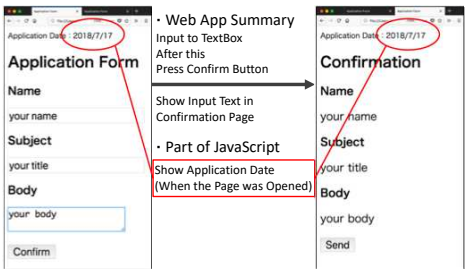

Fig. 9. Created web application

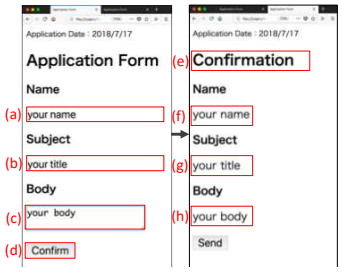

Fig. 10. Screen status

5.2 Testing with Selenium

We implement our proposed method using a tool for testing web applications called Selenium [5]. This test tool is widely used for automating the test process. In Selenium, each test case is described as a sequence of test procedures such as "Receive the input from a keyboard → generate a webpage → compare the generated page with the teacher data" in an auto-executable program code. Furthermore, it is easy to execute arbitrary application programs on Selenium and send the data obtained during the test procedures to those other application programs.

5.3 Test Process for Target Web Applications

The purpose of the test process is to ensure that the web application is implemented according to the specification. Therefore, the test process for this target web application is composed of the followings: (Test for Spec. 1) in the initial webpage, it is tested to ensure that the page transition occurs successfully by entering a character string in the text boxes and pressing the confirmation button; (Test for Spec. 2) subsequently, in the transition destination page, it is tested to ensure that the character string "Confirmation" is displayed in the heading. If both tests pass, then the test result is "success", otherwise it is a "failure". Fig. 10 shows an example of the screenshot images taken during the test.

In Selenium, we need to describe a test case as a sequence of test procedures. For this target web application, the sequence of the test procedures is as follows:

(1) Launch the web browser

(2) Display the web application page
(3) Input string in textbox (a) of Fig. 10
(4) Input string in textbox (b) of Fig. 10
(5) Input string in textbox (c) of Fig. 10
(6) Press button (d) of Fig. 10
(7) Compare headline (e) of Fig. 10 with "Confirmation" to verify the page transition
(8) Compare Strings (f), (g), and (h) of Fig. 10 with texts that are inputted at step (3), (4), and (5), respectively, to verify whether the inputted strings are displayed.

An example of the test case description in the Python language for the above procedures is as follows:

```
browser.find_element_by_name('name').send_keys(”your name")
browser.find_element_by_name('title').send_keys("your title")
browser.find_element_by_name('content').send_keys("your body")
browser.find_element_by_id('submitBtn').click()
assert "Confirmation" in browser.page_source
assert "your name" in browser.find_element_by_id('name').text
assert "your title" in browser.find_element_by_id('subject').text
assert "your body" in browser.find_element_by_id('body').text
```

The first to the third lines are operations for the steps (3) to (5) in the above procedures, respectively. The fourth line is for step (6), the fifth line is for step (7), and the sixth to eighth lines are for step (8). If the test result is successful then nothing is returned, and if the test result is unsuccessful "Assertion Error" will be returned.

5.4 Automatic Whitelist Generation and Operation

As explained in Section 4.4, our examination-based whitelist can be generated through the test process that is carried out as the final stage of the web application development. We can implement the automatic whitelist generation function by slightly modifying the program module of Selenium so that the examination-based whitelist can be generated by extracting all the JavaScript sections from each HTML page source code generated during the test process.

Selenium can acquire the HTML page source code, including the JavaScript, at any stage in the test process. Therefore, it is possible to generate a whitelist by automatically extracting the JavaScript parts from the HTML page source code after the execution of step (7) of the test procedure. The automatically generated whitelist is stored in the web server and used in the operation of the proposed method. With such operations explained in Section 4.2, users can be protected against XSS attacks.

5.5 Results

It can be confirmed that the automatic whitelist generation is simple, at least when using Selenium as a test environment. All the information/data required for the whitelist generation can be obtained through the test process of the web application development, and it is not necessary to alter the original test process at the time of web application development. Furthermore, even in more complicated web applications that need to test multiple pages, the essence of the test process does not change significantly. From the above, the automatic whitelist generation through the test process of web applications proposed in this paper is less cumbersome for

developers, and can be claimed to be an effective and reasonable method in terms of its availability and utility.

6 Conclusions

This paper proposed an examination-based approach to generating whitelists automatically for the prevention of XSS attack. In this paper, we focused on the script structures that are examined in the test process and defined them as whitelist.

From the viewpoint of completeness, the examination-based whitelist guarantees to execute only scripts that were confirmed beforehand in the test process. From the viewpoint of soundness, since the software test is performed based on the specification document of each web application, this method can automatically generate a whitelist consistent with the specification. We implemented and evaluated the proposed method using the software test tool Selenium. The results confirmed the effectiveness of the proposed method.

In future work, we plan to improve our method by investigating script structures that can detect XSS attacks with higher efficiency and to develop concrete verification procedures for whitelist checking.

References

1. "jQuery" [Online], Available: http://jquery.com/, [Accessed: 05- Dec- 2017]
2. "Blog Tool, Publishing Platform, and CMS - WordPress," [Online], Available: https://wordpress.org/, [Accessed: 05- Dec- 2017]
3. J.D. Meier, A. Mackman, M. Dunner, S. Vasireddy, R. Escamilla, and A. Murukan, "Web application security fundamentals," (2013). [Online], Available: https://msdn.microsoft.com/en-us/library/ff648636.aspx, [Accessed: 28- Dec- 2017]
4. B. Lord, "All about the onMouseOver incident," (2010). [Online], Available: https://blog.twitter.com/official/en_us/a/2010/all-about-the-onmouseover-incident.html, [Accessed: 28- Dec- 2017]
5. "Selenium - Web Browser Automation," [Online], Available: https://www.seleniumhq.org/, [Accessed: 05- Dec- 2017]
6. T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," Proceedings of the 16th international conference on World Wide Web. ACM, (2007) 601-610
7. L.A. Meyerovich, and B. Livshits, "Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser," Security and Privacy (SP), 2010 IEEE Symposium on. IEEE, (2010) 481-496
8. S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," Proceedings of the 19th international conference on World Wide Web. ACM, (2010) 921-930
9. T. Kakuta, T. Ohtori, Y. Fujii, and N. Taniguchi, "A detection method of malware infections based on graylists," IPSJ SIG Technical Reports, 2014-CSEC-66-16, 2014, pp.1-7 (in Japanese)