A simple XSS scanner

Aleksandar Kanchev

ABSTRACT

Millions of people access the Web each day. It has become an integral part of our everyday life. However, except for social networks, online shops and entertainment, one can stumble upon some unpleasant surprises in this otherwise beautiful place called "the Internet". As creators of content on the Web, we, software engineers, and our work are the targets for many attacks. Some try to expose us by proving that their abilities in penetrating applications are better than our abilities to protect our software, others try to steal our users' personal information, and the rest just like to see the world burn.

Research shows that by far the greatest number of security vulnerabilities come from Cross-site scripting (XSS). This paper describes a methodology for discovering XSS vulnerabilities in HTML-based applications and compares this method to existing solutions.

1. INTRODUCTION

As described by Wikipedia: "Cross-site scripting (XSS) is a type of computer security vulnerability typically found in web applications. XSS enables attackers to inject client-side scripts into web pages viewed by other users. Cross-site scripting carried out on websites accounted for roughly 84% of all security vulnerabilities documented by Symantec as of 2007. Their effect may range from a petty nuisance to a significant security risk, depending on the sensitivity of the data handled by the vulnerable site and the nature of any security mitigation implemented by the site's owner. "

The quotation above suggests that XSS vulnerabilities are a serious threat to modern web applications. This begs the question "Why is this such a big problem?". Is it that web developers are simply not aware of this threat or it is that difficult to find such security holes? A quick Google search shows us that there are many tools that claim to be good at finding XSS vulnerabilities. Even free ones that we can use online. Thus, it is not really that people are not aware of the threat. The fact that such issues still make for the majority of computer attacks shows that the existing scanners do not make a particularly good job of finding XSS-related issues.

1.1 Existing Solutions

Modern vulnerability scanners are known to report a lot of false positives and miss some important issues. This causes much frustration in application developers, often imposing the need for additional manual testing or even worse - leading to a complete negligence in terms of security.

Existing XSS vulnerability scanners are also primarily API-based. This means that they will first try to discover all endpoints exposed by the application. Then, using some finite list of popular exploits will try to inject some malicious content into our request processing logic.

Yet another downside of existing vulnerability scanners is that they are mostly developed by large enterprise companies. As a result, their code is proprietary, they are paid and tend to cost a fortune. Something that the average web developer cannot afford.

The purpose of this paper is to present a radically different approach to discovering XSS vulnerabilities in web applications.

2. METHOD

In contrast to existing XSS vulnerability scanners, this new methodology tackles the problem from a different perspective. It focuses on a type of XSS attacks called DOM-based XSS. As the name suggests, this type of XSS vulnerabilities happen when we modify the Document Object Model of an existing web application.

The proposed methodology consists of a smart bot that works inside the browser, in parallel to our application.

At a high level, we can consider the bot working in the following 5 stages:

1. The bot reads the source of the current page.
2. Finds all text inputs.
3. For each input found by (2), inserts malicious content.
4. Submits the content.
5. Checks if the content was executed.

This simple execution plan, raises a few questions. The following section tries to answer the most common ones.

3. DISCUSSION

Q: How does the bot have access to the page's source and how does it find all text inputs?

A: To have access to the page's source, the code for the smart bot, needs to be loaded and executed in the same browser instance and context as the application code. There are several ways to achieve that, some of which are:

- Loading the bot script from the HTML document representing the page to scan.
- Installing the bot as a browser plugin whose code is executed on demand or every time we load a page within the domain we want to scan.

Q: How does the bot make sure that it has found all existing inputs? Certain JavaScript frameworks can define custom HTML elements that represent inputs which can be misleadling.

A: This is a delicate area. However, since browsers follow a predefined web standard, they would not trust just any custom HTML tag. Even if they did, they will transform it to something the can work with. So, when the smart bot asks the browser for all inputs, it will most likely get the full list rather than not. Nevertheless, this topic is certainly worth investigating more and is considered as future work.

Q: Inputs can exist outside a form. How does the bot know how to submit the input?

A: This is also a tricky. However, usage of inputs in web applications most often follow a common pattern and are limited to a finite set of possible implementations. Covering all of them guarantees the scanner full coverage.

Q: How does the bot know if the malicious content was executed or not?

A: Since the bot can control the contents of the malicious script, it only inputs scripts whose output is easy to verify. For example, overriding the default implementation of the **window.alert** browser

function to send the details about the vulnerable input back to the bot.

4. RESULTS

In parallel with this new method for discovering XSS vulnerabilities, a proof of concept was developed for the smart bot. The prototype represents a Chrome browser extension that triggers a scan when we load a page inside the domain to scan, or on demand. Since the extension works in the same browser instance as the application to test, it has full access to the source of the pages it loads. The smart bot implementation does not rely on any JavaScript frameworks to find and submit the inputs but instead uses standard browser APIs. This ensures compatibility with all browsers and full functionality on any web page that can be rendered by the browser.

To test this smart bot implementation, three separate web applications were developed. All applications were developed in pure HTML and JavaScript. They all represent a student registration website, consisting of a form view where the users enter their data and a display view, that shows all registered students. To test the reliability of the smart bot, one of the applications sanitizes the user input to prevent any malicious content, while the other two do not.

The results for the smart bot prototype show that it was able to discover all unprotected inputs within the two vulnerable applications 100 percent of the time, while it did not report any vulnerabilities for the application that does sanitize the input data.

The same applications were also scanned using a popular and widely used security scanner – IBM's AppScan. The results were quite different. AppScan did not report any vulnerabilities for our two unprotected applications. And it is not because AppScan does not scan for DOM-Based XSS. In AppScan's job configuration there is a separate switch to enable the so-called JavaScript Security Analyzer. When one opens the help menu for this configuration option it reads: "Select this option to perform static JavaScript analysis to detect a range of client-side issues, primarily DOM-Based Cross Site Scripting."

5. FUTURE WORK

The prototype developed as a proof of concept for the proposed methodology satisfies the initial requirements. However, there is further room for improvement. Below are listed the top items needed to take this idea from the prototype to the product level:

1. **More robust discovery and reporting mechanisms** – even though the scanner proves to work quite well so far, there are undoubtedly some yet-to-be-discovered corner cases that should be handled.
2. **Full automation** – currently the scanner needs to be manually triggered, once we navigate to the page we want to test. The fully automated version of such a scanner should take the URL of the application as an input and get back with a thorough report of what's vulnerable. This can easily be achieved by integrating the scanner with an already existing E2E test automation. The test automation is ideal for this use case, as it already knows how to navigate around the application. It also knows where all the forms, inputs, etc. are and how to validate if the user input was correctly processed. We just need to hook into such existing automation and replace the "good" data with malicious one.
3. **Integration with JavaScript frameworks** – currently the scanner works only with applications written or compiled to pure HTML and JavaScript. Even though those make for quite a large number, there are applications developed using various JavaScript frameworks. Such applications often define custom HTML tags or create a virtual DOM. To make this scanner fully support such applications, we should consider the specifics of the frameworks in question. For example, AngularJS introduces a form of expressions where anything inside the so called double-curly braces (e.g. **{{ foo.bar() == bar.baz() }}**) is also evaluated by the browser. As such, AngularJS expressions are also known to be a target for XSS attacks.

6. CONCLUSION

In conclusion, the proposed methodology has proven very reliable in finding XSS vulnerabilities in HTML-based web applications. Some more work needs to be put to add support for all popular frameworks for developing web applications. However, the essence of such work would require familiarizing the bot with the specifics of each framework rather that inventing a completely new approach for discovering vulnerabilities.

7. REFERENCES

[1] Chon, K. W.-G. (2003, January). Search Security Tech Target. Retrieved from TechTarget: http://searchsecurity.techtarget.com/Web-application-security-scanners-How-effective-are-they

[2] Cross-site scripting. (2017, May 31). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Cross-site_scripting

[3] Symantec. (2008, April). Symantec Internet Security Threat Report.

[4] IBM. (n.d.). *How JavaScript source code analysis works*. Retrieved from IBM Knowledge Center: https://www.ibm.com/support/knowledgecenter/en/SSW2NF_9.0.1/com.ibm.ase.help.doc/topics/c_javascript_analysis.html