Detection of DOM-based Cross-Site Scripting by Analyzing Dynamically Extracted Scripts

Suman Saha[1], Shizhen Jin[2,3] and Kyung-Goo Doh[3⋆]

[1] LIP6-Regal, France
Suman.Saha@lip6.fr
[2] GTOne, Seoul, Korea
jinshzh@gmail.com
[3] Hanyang University ERICA, Ansan, Korea
doh@hanyang.ac.kr

Abstract. A malicious hacker may inject untrustworthy payload in a dynamically generated page intentionally. If a web server does not adequately sanitize the input data, the inadvertent execution of client-side scripts injected by malicious users creates security problems. DOM-based Cross-site Scripting (XSS) is a type of XSS that creates such types of security problems in client side. This paper presents a static taint analysis for detecting DOM-based XSS holes from dynamically generated error pages, which directly addresses the absence of built-in filter function. We provide a measurement study that sheds light on the DOM-based XSS holes present in web applications and reveals the severity of this type of XSS in the web world. To the best of our knowledge, there is no directly related work on analyzing HTML pages for detecting DOM-based XSS holes and measuring study of the holes from huge number of web applications.

Key words: software security, DOM-based cross-site scripting, static analysis, web application security, scripts

1 Introduction

Improper validation on user input before returning to the client's web browser is one of the main reasons of Cross-Site Scripting (XSS) vulnerabilities. A malicious programmer may inject untrustworthy payload in a dynamically generated page intentionally. If a web server does not adequately sanitize the input data, the inadvertent execution of client-side scripts injected by malicious users creates security problems, letting an attacker easily circumventing the same-origin policy [1].

⋆ Corresponding author. This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2012-000046), and (Grant 0004-20211) Business for Cooperative R&D between Industry, Academy, and Research Institute funded by Korea Small and Medium Business Administration in 2011.

DOM-based Cross-Site Scripting(XSS) [2] is an XSS vulnerability existing within client-side pages. For instance, if any URL request parameter is accessed to write its information in the HTML body or perform any DOM-based operation without validating, a DOM-based XSS hole will likely be present, since this written data will be re-interpreted by browsers as an HTML document that could include additional client-side scripts. Instead of sending data to server, this XSS might modify DOM environment in the client side to exploit. Hence, the server stays out of scope to handle this sort of cross-site scripting. As a consequence, validation on user input in the server side cannot stop DOM-based XSS exploiting. To handle DOM-based XSS, we need to concentrate on client-side HTML pages rather than server-side pages.

Say that the content of `http://www.vulnerable.site/welcome.html` is as follows:

```
<html>
  <title>Welcome!</title>
  <body>
    Your name is:
    <script>
      var pos = document.URL.indexof(name=)+5;
      var name = document.URL.substring(pos, document.URL.length);
      document.write(name);
    </script>
  </body>
</html>
```

The user input is not sanitized and later used to write in the HTML body. The script embedded in the page works alright when the user input is an expected string, i.e., an identifier, as follows:

```
http://vulnerable.site/welcome.html#name=Suman
```

However, if the user injects any malicious script code as follows:

```
http://vulnerable.site/welcome.html#
       name=<SCRIPT>alert(document.cookie)</SCRIPT>
```

the script code is executed at the client side, which is obviously not the developer's intention. The string beyond `#` sign is a fragment, i.e., not part of an actual query, that is not sent to server. Therefore, the server will get only a query string without the malicious part of the input data. As a result, many strong XSS filters at server-side do not even recognize such attacks. Since most of detectors and firewalls ignore client-side pages, they are inherently not able to detect DOM-based XSS.

In this paper, we present the method of detecting DOM-based XSS holes from HTML pages, which directly addresses the absence of applying sanitization functions to user input. The detection method proposed in this paper first induces the server to generate pages by sending dynamic requests, then extracts

script codes from the pages, and then performs taint analysis to decide whether or not the user input is sanitized before it reaches the critical hot spot. We provide a measurement study that sheds light on the DOM-based XSS holes present in the web applications and reveals the severity of this type of XSS in the web world. To the best of our knowledge, there is no directly related work on analyzing HTML pages for detecting DOM-based XSS holes and measuring study of the holes from huge number of web applications.

The rest of this paper is structured as follows. Section 2 presents how to detect DOM-based XSS. Finally, Section 6 concludes. The full version of this paper will include the experiment evaluation of the proposed system and related works.

2 Methodology

The presented work is targeted at the static detection of taint-style vulnerabilities. Tainted data are originated from possibly malicious user's inputs (called sources), and may cause security problems at some sensitive points in a program (called sinks). Applying a set of suitable filtering operations, tainted data can be untainted (sanitized), removing its harmful properties before reaching sinks. The goal of our analysis is to determine whether or not tainted data reaches sensitive sinks (any write method or any DOM-based operation) without proper sanitization. For this, we apply the technique of data-flow analysis, which is a well-understood topic in computer science and has been used in compiler optimizations for decades [3–5].

The URL parameter that is the major source of DOM-based XSS vulnerabilities comes from `document.location`, `document.URL`, or `document.referrer`. The `document` object represents the entire HTML document and can be used to access all elements in a page. `location`, `URL`, and `referrer` are sub-objects of the `document` object. The `document` object is a part of the `window` object and is accessed through the `window.document` property. In the code below, `document.location.href` acquires the current location of URL, and then `document.write` method writes that URL information in the HTML body.

```
document.write(You are using IE and visiting site
               + document.location.href + .)
```

Moreover, reference to DOM objects that may be influenced by the user (attacker) should also be inspected [2], including (but not limited to):

1. `document.URLUuencoded`
2. `document.location` (and many of its properties)
3. `window.location` (and many of its properties)

These are sources for URL information. Note that a `document` object or a `window` object property may be referenced syntactically in many ways explicitly through `window.location` or implicitly through `location`. Special concentration should

be given to patterns wherein the DOM is modified, either explicitly or potentially, either via raw access to the HTML or via access to the DOM itself.

Amit Klein gave in his report a list that should be considered when developing tool for detecting DOM-based cross-site scripting [2]. To write in HTML body, a developer may use any of followings that are the sinks for URL information:

- `document.write()`
- `document.writeln()`
- `document.body.innerHTML =`

The proposed DOM-based XSS detector first parses a JavaScript embedded HTML input file with respect to the combined grammar of HTML and JavaScript, and constructs its abstract syntax tree (AST). The JavaScript portion of the AST is then transformed into a linearized form resembling three-address code [5], and kept as a control flow graph. Then the detector analyzes data from source (`location`, `URL`, `referrer`) to sink (any `write` method or DOM-based operations). It observes whether or not the URL information is sanitized before reaching sink. Finally, it gives a message whether or not the given HTML page is vulnerable.

The DOM-based XSS detector does taint analysis that looks for only built-in specified filter function (e.g., `urlEncode` [16]) while it does data flow analysis from source to sink. In this phase, the detector at first locates sources, i.e., URL parameters, and then follows the program's data-flow from each source to sinks to check if the value of the source is sanitized by specified built-in filter function before reaching sinks.

3 Conclusion

Cross-Site Scripting is one of the most rising vulnerabilities found in modern Web application. We provide a detection method and a first measurement study on DOM-based XSS holes. Specific security systems (such as firewalls, software proxies, etc.) that are external to the application may be unsatisfactory for several reasons. Rather, a web application should be intrinsically secure, by adapting secure programming practices, in order to preserve its invulnerability. DOM-based XSS is the result of insecure practice of JavaScript engineering. This paper suggests analyzing client-side HTML pages along with server pages to detect whole range of XSS attack. In the future, we are planning to employ the semantics of filtering function with static-taint analysis to detect DOM-based XSS in HTML files.

References

1. JavaScript Security: Same Origin. Mozilla Foundation (2006)
2. Klein, A.: DOM-based Cross-Site Scripting of the Third Kind (2005) http://www.webappsec.org/projects/articles/071105.html

3. Aho, A. V., Sethi, R., Ullman, J. D.: Compiler: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA (1986)
4. Muchnick, S. S.: Advance Compiler Design and Implementation. Morgan Kaufmann (1997)
5. Nielson, F., Nielson, H. R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York (1999)
6. Xie, Y., Aiken, A.: Static Detection of Security Vulnerabilities in Scripting Languages. In: 15th USENIX Security Symposium, pp. 179-192 (2006)
7. Huang, Y.-W., Hang, F., Yu, C., Tsai, C. H., Lee, D., Kuo, S.Y.: Securing Web Application Code by Static Analysis and Runtime Protection. In: 13th International Conference on the World Wide Web (2004)
8. Livshits, V.B., Lam, M.S.: Finding Security Errors in Java Programs with Static Analysis. In: 14th USENIX Security Symposium, pp. 271-286 (2005)
9. Jovanovic, N., Kruegel, C., Kirda, E.: Precise Alias Analysis for Syntactic Detection of Web Application Vulnerabilities. In: ACM SIGPLAN Workshop on Programming Language and Analysis for Security (2006)
10. Yue, C., Wang, H.: Charactering Insecure JavaScript Practice on the Web. In: 18th International Conference on the World Wide Web, ACM Press (2005)
11. Moshchuk, A., Bragin, T., Gribble, S. D., Levy, H. M.: A Crawler-based Study of Spyware in the Web. In: 13th Annual Network and Distributed System Security Symposium (2006)
12. Provos, N., Mavrommatis, P., Rajab, M. A., Monrose, F.: All Your iframe Point to Us. In: 17th USENIX Security Symposium (2008).
13. Wang, Y.-M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., King, S. T.: Automated Web Portal with Strider Honeymonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In: 13th Annual Network and Distributed System Security Symposium (2006).
14. Joo, B.-G., Min, B.-W., Chang, M.-S., Ahn, C.-K., Yang, D.-H.: Development of Vulnerability Scanner using Search Engine. The Journal of IWIT (The Institute of Webcasting, Internet and Telecommunication). Vol.9, No.1, pp.19–24 (2009).