Watch What You Write: Preventing Cross-Site Scripting by Observing Program Output

Matias Madou, Edward Lee, Jacob West and Brian Chess

Fortify Software
2215 Bridgepointe Pkwy, Suite 400
San Mateo, CA, 94404
{mmadou, elee, jacob, brian}@fortify.com

Abstract. We introduce a dynamic technique for defending web applications that would otherwise be vulnerable to cross-site scripting attacks. Our method is comprised of two phases: an attack-free training period where we capture the normal behavior of the application in the form of a set of likely program invariants, and an indefinite period of time spent in a potentially hostile environment where we check to make sure the application does not deviate from the normal behavior. We demonstrate that our approach is both effective at protecting vulnerable applications and capable of doing so without introducing a prohibitive amount of overhead. Our experiments suggest that this invariant-based technique is the most powerful and accurate automated mechanism for identifying and protecting against the widest range of cross-site scripting vulnerabilities.

1 Introduction

Cross-site scripting (XSS) is the most wide-spread vulnerability in web applications today. The 2007 update to the OWASP Top 10 ranks XSS as the #1 web application security vulnerability [5] and data from the MITRE Common Vulnerability Enumeration (CVE) project show that the rate of publicly disclosed XSS vulnerabilities is increasing [2]. These data support the idea that XSS vulnerabilities are both easy for programmers to introduce and easy for attackers to find, which suggests that a technique for defending vulnerable applications at runtime would be a boon to web security.

An XSS vulnerability permits attackers to include malicious code in the content a web site sends to a victim's browser. The malicious code is typically written in JavaScript, but it can also include HTML, Flash or any other type of code that will be interpreted by the browser. Attackers can exploit an XSS vulnerability in a number of different ways. They can steal authentication credentials, discover session identifiers, capture keyboard input, or redirect users to other attacker-controlled content [4].

The best approach to preventing XSS vulnerabilities is a programmatic combination of input and output filtering: validate all input using a whitelist to ensure it contains only expected values and validate output bound for the web browser to ensure that it does not contain malicious code [1]. However effective,

such a solution requires a concerted commitment to preventing XSS vulnerabilities and is often difficult to implement consistently, particularly in legacy programs that were developed without security in mind. Some Web Application Firewalls (WAFs) implement less effective protections for XSS vulnerabilities that often focus on identifying possible attacks using input filtering at the network or web server layer. These solutions suffer from wide-spread false negatives (missed attacks) and false positives (warnings raised during normal behavior) because they lack the necessary application context to determine which data represent a feasible attack and which do not [6, 13].

This paper introduces a method for defending web applications against XSS vulnerabilities at runtime using fine-grained dynamic output inspection. The primary difference between our approach and other automated techniques for mitigating the danger posed by XSS vulnerabilities at runtime is that we identify dangerous values as they are written into the HTTP response rather than as they enter the program. This enables us to defend against attacks that cannot be witnessed at the HTTP request level, such as attacks that rely on data that are batch loaded into a database, arrive via web services or another non-HTTP entry point, or that appear in an encoded form when they enter the program. Inspecting output rather than input also enables us to implement more fine-grained protections that better model real-world programming scenarios where certain dynamic behavior is acceptable in some situations but not in others. Finally, inspecting output as it is sent to the user means that not only do we identify attacks, but when a likely invariant is violated we are able report a true XSS vulnerability in the application because the malicious data have reached the user.

The remainder of the paper is organized as follows. Section 2 introduces our approach. Section 3 provides experimental results that compare the effectiveness of our approach with a popular web application firewall. Section 3.4 discusses the challenges and limitations that face our approach. Section 4 discusses related work, Section 5 mentions ideas for future work, and Section 6 summarizes our conclusions.

2 Method

An XSS vulnerability can take one of three forms. *Reflected XSS* occurs when a vulnerable application accepts malicious code as part of an HTTP request and immediately includes it as part of the HTTP response. *Persistent XSS* occurs when a vulnerable application accepts malicious code, stores it, and later distributes it in response to a separate HTTP request. *DOM-based XSS* occurs when the malicious payload never reaches the server–it is only seen by the client [7].

Our approach defends web applications against reflected and persistent XSS attacks. It works in two phases. In the first phase we monitor the target application during an attack-free training period with a finite duration and generate *likely invariants* on normal program behavior. The likely invariants are conditions that always hold during the training period. They are all related to the

types of output the program writes to the HTTP response. We expect this phase could be carried out in conjunction with typical functional testing, which is intended to exercise a wide range of normal program behavior. The likely invariants we derive are guaranteed to hold only for the program behavior observed during the training period, but if the program is well exercised during the training period, the invariants we derive are likely to be ones that programmers believe will always hold. Once we have developed a set of likely invariants, we monitor the application when it is deployed in a production environment. We report a problem when we identify program behavior that violates one or more likely invariants.

In this section we give an example of the kind of vulnerability we aim to defend against and detail the two phases of our approach.

2.1 Example

Imagine a simple blogging application. The blog contains a page that allows a user to submit the title and body of a new blog entry. An HTTP request to add a new entry is handled by the application server, which dispatches the request to the preview page named newblog.jsp. The source for newblog.jsp includes the following code:

```
<tr>
  <td class=newsCell><%= element.getTitle() %></td>
  <td class=newsCell><%= element.getBody() %></td>
</tr>
```

The URL portion of a typical HTTP request for this page might look like this:

```
http://example.com/preview.do?title=First&body=I+got+here+first.
```

in which case the page will generate the following HTML output as part of the HTTP response:

```
<tr>
  <td class=newsCell>First</td>
  <td class=newsCell>I got here first.</td>
</tr>
```

Another typical URL might look like this:

```
http://example.com/preview.do?title=Me&body=
My+photo%3A+%3Cimg+src%3D%22me.png%22%2F%3E
```

which will generate the following output:

```
<tr>
  <td class=newsCell>Me</td>
  <td class=newsCell>My photo: <img src="me.png"/></td>
</tr>
```

This page is vulnerable to reflected XSS. For example, if an attacker requests the the URL

```
http://example.com/preview?title=XSS&body=
%3Cscript%3Ealert('vuln+to+xss')%3C%2Fscript%3E
```

the page will generate the following response:

```
<tr>
  <td class=newsCell>XSS</td>
  <td class=newsCell><script>alert('vuln to xss')</script></td>
</tr>
```

When a browser renders this HTML, it will execute the JavaScript within the script tag.

2.2 Likely Invariant Generation

An *invariant* is a property that always holds at a certain point in a program. Programmers sometimes check important invariants with assert statements or other forms of sanity checking logic. In order to determine likely invariants related to XSS, we insert monitors into the program that record values included in content written to the HTTP response.

We define an *observation point* to be a method call that writes directly to the HTTP response. These are the locations we will characterize and monitor for XSS attacks.

The JSP code from newblog.jsp in Section 2.1 could be translated into the following Java code:

```
20:  out.write("<td class=newsCell>");
21:  out.print(element.getTitle());
22:  out.write("</td>\t\r\n    <td class=newsCell>");
23:  out.print(element.getBody());
24:  out.write("</td>");
```

It contains five observation points. Before the training period we re-write the program's bytecode to insert monitors around these method calls. We use a simple static analysis of the program to avoid monitoring method calls that can only write static content to the HTTP response because they are trivially immune to XSS vulnerabilities. For the code above, the relevant observation points are the calls to javax.servlet.jsp.JspWriter.print(String s) on lines 21 and 23, because they are the only two methods that write dynamic content to the HTTP response.

We define an *observation context* to be the state of the program when an observation point is invoked. We represent the observation context with the URL from the HTTP request and the current call stack. Although we only track the URL and call stack, it is possible to track other state information such as HTTP request parameters, HTTP request headers, or user roles. In general, the

more dimensions there are to the observation context, the more fine-grained and robust the likely invariants and detection algorithm will be. By keeping track of contexts rather than just observation points, we can develop a different set of likely invariants for each context in which an observation point is used.

When an observation point executes, we examine what we know about the associated context. If we have not seen the context before, we use the argument to the observation point method to establish a set of likely invariants. If the context already has likely invariants associated with it, we check to see if any of the likely invariants are violated by the current method argument. If a likely invariant is violated, we update the likely invariant to make it consistent with the new behavior.

In our current implementation, all likely invariants are of the form *The substring S always occurs X times at this observation point.* We choose substrings which consist of patterns that could be part of an XSS attack, such as *<script*, *<img* and *javascript:*. Our collection of patterns comes from the XSS attacks we have studied. Counting the number of occurrences of each pattern allows us to establish a baseline of expected behavior. After the training period, any deviation from the expected behavior is considered a violation of the likely invariant.

Consider the application of this technique to the two normal requests for newblog.jsp given earlier. Although our implementation considers a wide range of relevant substrings, for simplicity sake consider only the following values for this example:

```
<script
<img
javascript:
```

If the two requests are the extent of the training data, we will establish the following likely invariants:

```
line 21: The substring "<script" always occurs 0 times
line 21: The substring "<img" always occurs 0 times
line 21: The substring "javascript:" always occurs 0 times

line 23: The substring "<script" always occurs 0 times
line 23: The substring "javascript:" always occurs 0 times
```

The invariants for line 23 will allow an image tag but will not allow an attribute that contains the string *javascript:*. This preserves the intended functionality of the application while preventing a popular form of XSS attack. (Other patterns are required in order to prevent other XSS varieties.)

For ease of understanding, we have labeled each invariant as corresponding to either line 21 or line 23, but the observation context also includes the URL and a call stack. This distinction has not been important in the examples given thus-far, but it is critically important for establishing likely invariants when the same method call can be invoked from more than one place in the program. Consider the following modified version of the JSP code from newblog.jsp that

uses the <logic:iterate> and <bean:write> tags to output the title and body values:

```
<logic:iterate id="element" name="profiles" scope="request"
               type="com.blog.postnew" >
  <tr>
    <td class=newsCell>
      <bean:write name="element" property="title"/></td>
    <td class=newsCell>
      <bean:write name="element" property="body"/></td>
  </tr>
</logic:iterate>
```

This JSP code will be transformed into the following Java code:

```
20:  WriteTag jsp_beanwrite_title;
21:  jsp_beanwrite_title.setName("element");
22:  jsp_beanwrite_title.setProperty("title");
23:  jsp_beanwrite_title.doStartTag();
     ...
30:  WriteTag jsp_beanwrite_body;
31:  jsp_beanwrite_body.setName("element");
32:  jsp_beanwrite_body.setProperty("body");
33:  jsp_beanwrite_body.doStartTag();
```

Notice that the code does not directly invoke the methods responsible for writing the dynamic output to the HTTP response. The call to javax.servlet.jsp.JspWriter.print() is hidden within the implementation of doStartTag(), which is invoked from two distinct program points at line 23 and line 33. In order to establish different sets of likely invariants for the two calls, we must take the call stack into account.

2.3 Runtime Monitoring

When the program runs in a production environment, we insert monitors at method calls used to write values to the HTTP response and use static analysis to avoid monitoring method calls that only write static content. This time the monitors check observed behavior against the likely invariants derived during the training period. When a likely invariant is violated, the monitor takes one of several actions, which include logging the attack or raising an exception. The program's owner can configure the monitors to take an action appropriate for the program and execution environment in question.

When a monitor executes in production, we identify the likely invariants to apply by matching the current program state with the observation contexts witnessed during the training period. Comparing the entire call stack is costly in terms of overhead. To avoid doing so, we compute a minimal set of call stack nodes during the training period that uniquely describe a group of contexts that

share the same likely invariants. To compute this minimal set, we first group contexts that shared the same likely invariants. Then, for each call stack in each of the group, we compare the last node before the observation point with the node in the corresponding position in call stacks for other groups. If the node is unique, then we continue comparing the remaining contexts in the current group. If the node is not unique, then we begin a breadth first search to find a node or set of nodes that are unique. If no single node position uniquely differentiates the call stacks in one group from all others, then we expand our scope to two nodes and so on until this requirement is met.

Checking likely invariants independently is conceptually simple but computationally expensive. We speed up the checking at runtime by building regular expressions out of the likely invariants for each observation point, which reduces the overall number of comparisons performed. A set of special substrings can be combined into a single regular expression if the likely invariants associated with them all require zero occurrences of the substrings. Given a training period comprised of the normal request given in Section 2.2, the invariants can be combined without loss of accuracy as follows:

```
line 21: The regular expression
           "(<((img)|(script))|(javascript:)" matches 0 times
line 23: The regular expression
           "(<script)|(javascript:)" matches 0 times
```

3 Experimental Results

In order to measure the effectiveness of our approach, we conducted an experiment to compare our approach against ModSecurity [1] (v2.5.1), a popular web application firewall, with only the latest rules (v1.6.0) related to XSS enabled. In this section we outline the details of the experiment and show the results of our comparison between the two tools in terms of their effectiveness at protection a vulnerable application and the overhead they introduce.

3.1 The Setup

For our experiment, we selected Pebble (v2.0.2), which is a lightweight, open source blogging application implemented in Java that we feel is representative of the kind of software that stands the benefit from our technique. The application contains multiple XSS vulnerabilities, which we reported to the project's maintainers. Specifically, our experiment focused on an XSS vulnerability caused by insufficient validation of the *name* parameter, which stores the name of the current user and is included in various pages.

The environment for our experiment consists of client machine running Microsoft Internet Explorer (v6.0) that interacts with a server running Apache

[1] http://www.modsecurity.org/

httpd (v2.2.8) in front of Apache Tomcat (v5.5.25). We selected Internet Explorer 6.0 for our client browser because it is known to be extremely vulnerable to XSS. Although IE6.0 is an old browser, in March 2008 30% of users still rely on IE6.0 [2]. Pebble does not require Apache httpd, however, we chose this setup because ModSecurity installs as an Apache httpd module and because this corresponds with real-world deployment scenarios. This setup is illustrated in Figure 1.

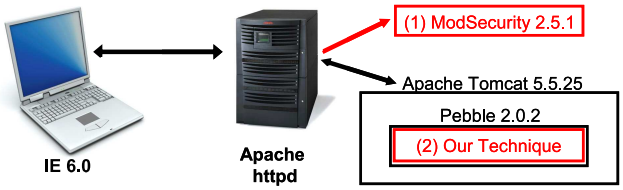

Fig. 1. Experimental Setup

3.2 Security

To evaluate the effectiveness of the two tools at preventing attacks against the vulnerable use of the *name* parameter, we consider the following four attack strings:

1. <script>alert('vulnerable to XSS')</script>
2.
3.
4.

where the spaces in the last two attack strings are tab (HT) characters.

With only the vulnerable version of Pebble running, all four attacks above trigger a pop-up in IE6.0 that contains the string 'vulnerable to XSS' on each page that displays the *name* parameter, thus exploiting the XSS vulnerability.

To evaluate our approach, we first developed an automated series of requests that exercises the normal behavior of Pebble and provide an attack-free training period. Specifically, during the training phase we modified user details, added two static pages, two blog entries and visited a handful of additional pages. From this interaction, we formed a set of likely invariants.

Next we tested Pebble with each of the attack strings; once with ModSecurity installed on the Apache httpd server and once with our likely invariants applied to the application. ModSecurity successfully protected the application against three out of the four attacks. However, it failed to catch the fourth attack pattern and allowed us to successfully exploit the XSS vulnerability. Using the likely

[2] http://www.w3schools.com/browsers/browsers_stats.asp

invariants we developed during the training period, our approach was able to block all four attacks.

Next we asked ourselves whether ModSecurity could be customized to capture the fourth attack. One approach is to add a rule that blocks *<img*. However, with this rule in place ModSecurity would prevent users from posting images to blogs, which violates the design requirements of the application. A second approach is to add a rule that blocks variants of *alert*. However, this is likely to introduce false positives on content that contains the word *alert*. Our technique does not suffer these limitations because it restricts attackers from inserting malicious values in the *name* parameter without impacting the functionality of the rest of the application.

3.3 Overhead

To measure the overhead introduced by ModSecurity and our technique, we have setup three distinct scenarios for Pebble: (1) unprotected, (2) protected with ModSecurity (only XSS rules enabled) and (3) protected with our invariant-based approach. To conduct the test, we made an attack free reference set containing 6469 requests, of which 104 were *POST* and 6365 were *GET*. We executed this series of requests five times for each of the three test scenarios and took the middle three timings to compute an average overhead. ModSecurity introduced an average overhead of 0.5%, while our technique contributed an average overhead of 2.11%. We feel that both tools introduced an acceptable level of overhead for use in many production environments.

3.4 Discussion

Beyond the attacks mentioned in our experiment, we would like to describe two additional classes of attacks that distinguish our approach from WAFs, such as ModSecurity. First, consider an attack against the persistent XSS vulnerability described in Pebble that relies on the attackers ability to insert malicious values directly into the database. An attacker might accomplish this by inserting the value before the WAF is put in place, by including the attack string in data that are batch loaded into the database, or using another vulnerability, such as SQL injection. Once the attack string is in the database, a WAF will offer no protection because it does have access to the malicious value. In contrast, our approach performs equally well regardless of the attack vector selected because we identify and prevent the attack as it leaves the application.

Another scenario where our approach has a significant advantage over WAFs is in programs where input arrives in an encoded format and is later decoded by the application. If, for example, the *name* value in Pebble were expected in an encoded form, the WAF would not identify the attacks described above, despite the fact that the application would subsequently decode them, rendering them dangerous once again. Because our technique is applied at the last possible moment before potentially dangerous values exit the application, our view of the data most closely matches what the user, or victim in this case, receives.

Finally, when a likely invariant is violated, not only have we identified an attack, but we have also found an application vulnerability because the malicious data was allowed to pass through the application to the point where it is rendered to the user. Therefore, the results produced from our technique not only represent real attacks that have been thwarted, but also provide details of security vulnerabilities that can be passed back to the development team for remediation. In contrast, results produced by a WAF represent only attempted attacks, and do not relate to real vulnerabilities or provide any help in identifying any real vulnerabilities that might exist in the application.

The accuracy of likely invariants depends on the extent of normal program behavior exercised during the training period; normal program behavior that violates a likely invariant but is not witnessed during the training period will result in false positives when the invariant is later enforced. Conversely, the presence of attack data or normal program behavior that we cannot distinguish from attack data will introduce false negatives because we are unable to derive a likely invariant.

We are aware that it is unlikely that a given training period will exercise all possible permutations of normal program behavior, however, our research indicates that a training period that is sufficiently broad to avoid false positives is achievable in practice. With respect to false negatives, in a controlled environment it should be possible to ensure that no attack data are included in the training period.

One might argue that the proposed technique can also be used to protect against other types of attacks, such as SQL or command injection. In some cases we believe our approach is applicable. We recommend that programmers avoid SQL injection vulnerabilities by using prepared statements and bind variables instead. By using prepared statements, the control is separated from the data, and as such an attack is no longer possible. The XSS problem is fundamentally different from SQL injection because the XSS cannot be prevented by separating control from data. The proposed technique is suitable to protect against command injection attacks because, as with XSS, programming interfaces for executing system commands do not allow the programmer to separate control from data.

Unlike network based input filtering technology, our method only needs to account for variations of XSS patterns that will be interpreted directly by browsers rather than accounting for packet fragmentation attacks or server specific encoding and decoding. The variations that we currently account for include: opening tags, closing tags, null characters, JavaScript event handlers, variations of *javascript:*, CSS (Cascading Style Sheets) *import* and CSS *expression* directives[3]. However, if a new attack pattern is discovered for a popular browser, our patterns will need to be updated.

Currently, our implementation is limited to monitoring observation points that take string arguments. Methods that output characters or byte arrays will not be analyzed. This is not a limitation of our methodology, but rather a short-

[3] http://ha.ckers.org/xss.html

coming of our implementation. Enhancing our implementation to support non-string based output is planned in future work.

4 Related Work

Daikon uses dynamic analysis to identify likely invariants, such as a given variable always holding a constant value or being non-zero [3]. Daikon has been used to perform various tasks, including generating test cases, predicting incompatibilities in component integration, automating theorem proving, repairing inconsistent data structures, and checking the validity of data streams. We took inspiration from the Daikon work to produce our likely invariants.

Automatic discovery of XSS is often performed at runtime by penetration testing tools. However, these tools are dependent on their ability to effectively crawl the application under test and can have difficulty scanning applications where navigational links and content are controlled dynamically with JavaScript. Static source code analysis tools are effective at discovering XSS vulnerabilities and have the advantage of providing full code coverage, but also have difficulty with dynamically generated content. We believe a combination of runtime and static analysis techniques is the most effective solution for identifying XSS vulnerabilities [12].

While both runtime and static analysis tools provide effective detection of XSS vulnerabilities, both techniques rely on the programmer to remediate the issues. Our solution, which is designed to both identify and correct XSS vulnerabilities in software, is built on the foundation set forth in the presentation *Countering The Faults of Web Scanners Through Byte-code Injection* [10] and an associated paper [11]. This work outlines a method for using aspect-oriented programming to insert code to passively monitor and actively protect web applications.

ModSecurity is a web application firewall that monitors inbound and outbound traffic to actively identify XSS attacks, as well as attacks against against SQL injection, file injection and other vulnerabilities. Because ModSecurity and other web application firewalls inspect traffic in and out of web applications at the HTTP-layer, they report attacks instead of vulnerabilities and often lack the necessary application context to differentiate malicious activity from expected behavior.

Similarly, Snort [14] is an open source network-based intrusion prevention and detection system that can detect XSS attacks. Snort attempts to model normal program behavior in some situations, but does not apply this technique to the detection of XSS. LibAnomaly is a research project for anomaly detection developed by the Computer Security Group at UCSB [8, 9]. The anomaly detection system in this library analyzes web-based attacks by taking web server log files as input and produces an anomaly score for each request.

5 Future Work

The invariants we currently create are akin to a blacklist: they specify particular patterns that should not appear in the output when the program runs. We plan to add whitelist invariants too. *Whitelist* invariants are of the form *The argument string always matches the regular expression R.* We will choose regular expressions that match textual representations of common data types that are inert when rendered by a web browser. For example, we will have regular expressions for integers, email addresses, and phone numbers. A whitelist mechanism is particularly useful in accurately protecting against XSS vulnerabilities where an application includes attacker-controlled input in existing JavaScript content because none of the usual malicious strings are necessary to cause the code to be executed in this case.

A significant portion of the overhead introduced by our technique comes from pattern matching. Currently, our implementation uses the default *java.util.regex* with basic optimizations for the patterns itself. We would like to investigate single pattern matching algorithms and the multi-pattern matching algorithms that build upon these. Also, by using the previously mentioned whitelist, we might kill two birds with one stone. First, whitelisting is generally known to be better for protection than blacklisting. Second, it might reduce the overhead. It takes much longer for the engine to declare that a regular expression did not match an input string (blacklisting) than it does to find a successful match (whitelisting).

In order to make our technique more resilient to evolving program behavior and incomplete training data, we will investigate a mode where we can also derive and update invariants in production. This is challenging both because we can make no guarantees that the program behavior we observe will be free from attacks and because the performance constraints of a production system are very different from one in a testing environment, however, we feel optimistic that by targeting specific behavioral idioms we can make meaningful progress in this area.

Finally, the task of modeling normal program behavior would be simplified if we could accurately differentiate user input from application-controlled values in production systems. To this end, we will explore integrating dynamic taint propagation techniques, which has been used to identify security vulnerabilities related to the misuse of user input in applications under test, into our approach. With these capabilities, we could apply the techniques described in this paper selectively in situations where the data in question are user controlled, and avoid unnecessary effort on data that are under the application's control.

6 Conclusions

The best way to prevent XSS is for programmers to write code that makes successful attacks impossible. However, it is easy to introduce XSS vulnerabilities, and that makes it worthwhile to have an additional layer of defense. We

have described an automatic, dynamic XSS protection that works by taking a fine-grained approach to observing program output.

Our technique provides good protection with a minimal amount of overhead. In order to keep the overhead to a minimum, we explored a number of optimizations. For example, it is important to monitor only dynamically generated content and not static content. At the moment, our implementation monitors nearly all writes to the response. Only very specific cases are filtered out. A more sophisticated algorithm that better distinguishes between static and dynamic content might further improve performance.

Our experimental results indicate that our approach provides the most complete automated protection against XSS vulnerabilities to-date. We protect against a wider range of attacks than previous techniques that have been applied to this problem. Furthermore we do so with the added benefit of reporting real application vulnerabilities rather than just attempted attacks, which makes our technique an effective component of an iterative feedback loop designed to improve the overall security of the software in question.

Acknowledgments

References

1. Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley, 2007.
2. Steve Christey and Robert A. Martin. Vulnerability type distributions in cve, v1.1, May 2007. http://cwe.mitre.org/documents/vuln-trends/index.html.
3. Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, pages 35–45, Dec 2007.
4. Seth Fogie, Jeremiah Grossman, Robert Hansen, and Anton Rager. *XSS Exploits: Cross Site Scripting Attacks and Defense*. Syngress, 2007.
5. OWASP Foundation. The ten most critical web application security vulnerabilities: 2007 update, 2007.
6. Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *10th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.
7. Amit Klein. Dom based cross site scripting or xss of the third kind, Jul 2005.
8. C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proceedings of ESORICS*, Oct 2003.
9. C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communication Security (CCS '03)*, pages 251–261, Washington, DC, October 2003. ACM Press.

10. Toshi Kureha. Countering the faults of web scanners through byte-code injection. In *BlackHat Europe*, Apr 2007.
11. Toshi Kureha and Brian Chess. Countering the faults of web scanners through byte-code injection, Apr 2007. https://www.blackhat.com/presentations/bh-europe-07/Kureha/Whitepaper/bh-eu-07-chess-kureha-WP.pdf.
12. Edward Lee. Comparing application security tools. In *Defcon*, Aug 2007. http://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-lee.pdf.
13. Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. In *Secure Networks, Inc.*, Jan 1998.
14. Martin Roesch. Snort - lightweight intrusion detection for networks. In *LISA '99: Proceedings of the 13th USENIX conference on System administration*, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.