Stopping injection attacks with proof carrying code and structured data

Ville Tirronen

Abstract. Injection attacks top the lists of the most harmful software vulnerabilities. Injection vulnerabilities are both commonplace and easy to exploit, which makes developing injection protection schemes important. In this article, we show how injection attacks can be practically eliminated with the use of structured data paired with cryptographic verification code on transmission.

Keywords: XSS, Injection, SQL-injection, Proof carrying code, Cryptographic hash

1 Introduction

SQL injections are one of the most persistent and easy to exploit software security issues. Regardless of decades of effort to stop injections, these attacks are still a daily occurrence. Similarly, cross-site-scripting flaws top the web security issue lists. These two attacks are but small part of the whole family of injection attacks, which makes it important to develop general means for stopping them.

One effective way for stopping injection attacks is to understand that SQL queries, web pages, XML documents, and other injection targets *are not strings of characters*. They are only written as such when authored by a person, or serialized to such when transmitting them over the wire. Thus, a simple strategy for preventing injection attacks is to represent potential injection targets as abstract syntax trees (ASTs) and only manipulate them structurally. When expressions are represented as ASTs, modifying the content of a node (representing, say, a query parameter of an SQL-query) cannot modify the structure of the AST. That is, no new query clauses or script nodes can appear spontaneously.

However, injection vulnerabilities are often seen as input validation problems [10]. We find such a view problematic. Firstly, it seems that such input validation is extremely difficult to do correctly. For example Balzarotti et. al. [6] describe a tool that discovered multiple vulnerabilities from venerable sanitation schemes that had been "battle tested" for long time. Secondly, we claim that viewing SQL and other injection attacks as input validation difficulties we are trading an easy and well known problem (serialization) into a much harder problem, validation, for which no sufficiently powerful solution is known (cf. [12]).

Manipulating data as an AST guarantees that the data is always well formed and no classical injection attack can succeed. Unfortunately, there is an important caveat to this. The manipulated structures will eventually need to be serialized for

transmission and deserialized at the opposing end. There often are no guarantee that the deserialization is a strict inverse of serialization. Software components responsible for deserialization can belong to different vendor than the one doing the serialization. Additionally, some deserializers, the prime example being HTML-parsers, are intentionally lax in accepting input (cf. claims in [18]) as to better support human input.

That is, an input which is certainly well formed and free of injections before deserialization can be interpreted (possibly after several passes of transformations on the other end) in a quite a harmful way (see e. g., [11]). Thus, it seems that the only guarantee for safety obtainable by manipulating data in structure form is to first serialize and then deserialize the structure with the exact same parser as is used to finally interpret the data and to compare the result to the expectation. In the age of multiple different web browsers, dozens of different SQL databases and a horde of different programming language libraries, it requires little imagination to perceive how infeasible this would be in practice.

In this article we discuss a solution to injection attacks that occur due serialization/deserialization issues. We follow the general strategy of Proof Carrying Code (PCC) and obtain a minimally intrusive solution by forming structural preservation "proofs". To our knowledge, even though our injection prevention strategy arises from well known basic principles, it is novel (See the systematic survey by Hydara et. al. [12]).

2 Classical Injection Attacks and Syntax Trees

The term "injection attack" is often used loosely in the literature. To avoid terminological issues, we use the term "classical injection attacks" to refer to such injection attacks that modify the intended structure of the some expression that is later interpreted, and acted upon, by some software component. This definition covers much of the area what is understood as an injection attack. However, there are some attack types, such as *"return-to-JavaScript"* attacks [5], which are discussed under the title of injection attacks, but which do not actually change the expression structure.

To be clear, we only address classical, structure modifying, injection attacks in this article. For other type of injection attacks, different techniques are needed.

2.1 Abstract Syntax Trees

An Abstract Syntax Tree (AST) is a tree representation of the syntactic structure of computer language programs. Each node in the tree represents an operator of the said language, such as "`DIV`" in HTML or "`WHERE`" in SQL. The first step of analysis, or execution, of a computer language almost invariably consists of *parsing* the source code into an AST, upon which further analyses and transformations are performed.

For the purposes of this article, we consider an AST to be a labeled tree, with some limitations on the structure of the labels. Specifically, we expect that the

labels identify the operators, constants, literals, or variables which the AST node represents. All labels must be atomic, that is, without any internal structure.

Figure 1 shows an example of an AST generated from the SQL query '"`select * from logins where cookie='acf2..'`"'. In the Figure, `cookie`, `logins` and `Star` (ie., `*`) are considered variables, while `'acf2..'` is a literal. The `select` is a mixfix operator, the use of which is clarified by augmenting the AST with nodes `from` and `where`, each denoting specific cluster of arguments to `select`. Here, all labels considered are atomic, including the string literal `'acf2..'`. The literal might have structural meaning in other contexts, but as far as SQL statement is concerned, it is a string.

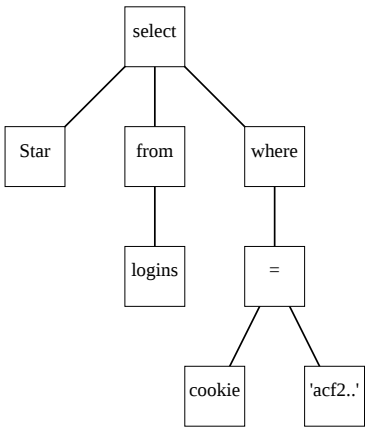

Fig. 1. An AST for"'select * from logins where cookie='acf2..'"'.

Direct manipulation of the AST inside a program is usually easier and less error prone than manipulating the serialized representation since the structure of the expression is explicit. That is, changing the structure of the expression must be done explicitly and will not happen as a side effect of some other change.

2.2 Attacks on AST based communications

Malfunctioning serialization/deserialization algorithms can expose the software system to injections even if each component in a distributed system manipulates expressions as proper structures. To give a concrete example, web browsers are intentionally lax in input they accept, taking the stance that giving user a possibly broken page is better than displaying nothing at all. For example, consider the following HTML snippet:

```
<img src="jav   ascript:alert('XSS');">
```

The snippet above contains a broken `src` attribute that the generating system has deduced harmless. However, a lenient browser may strip out the tab-character in the attribute, resulting in

```
<img src="javascript:alert('XSS');">
```

which causes a script to be executed in the browser. Here, the original attribute value could have been properly inserted into an AST, causing no structural changes. However, after being deserialized by a lenient parser it is converted to an injection attack. Figure 2 depicts the injection attack at AST level.

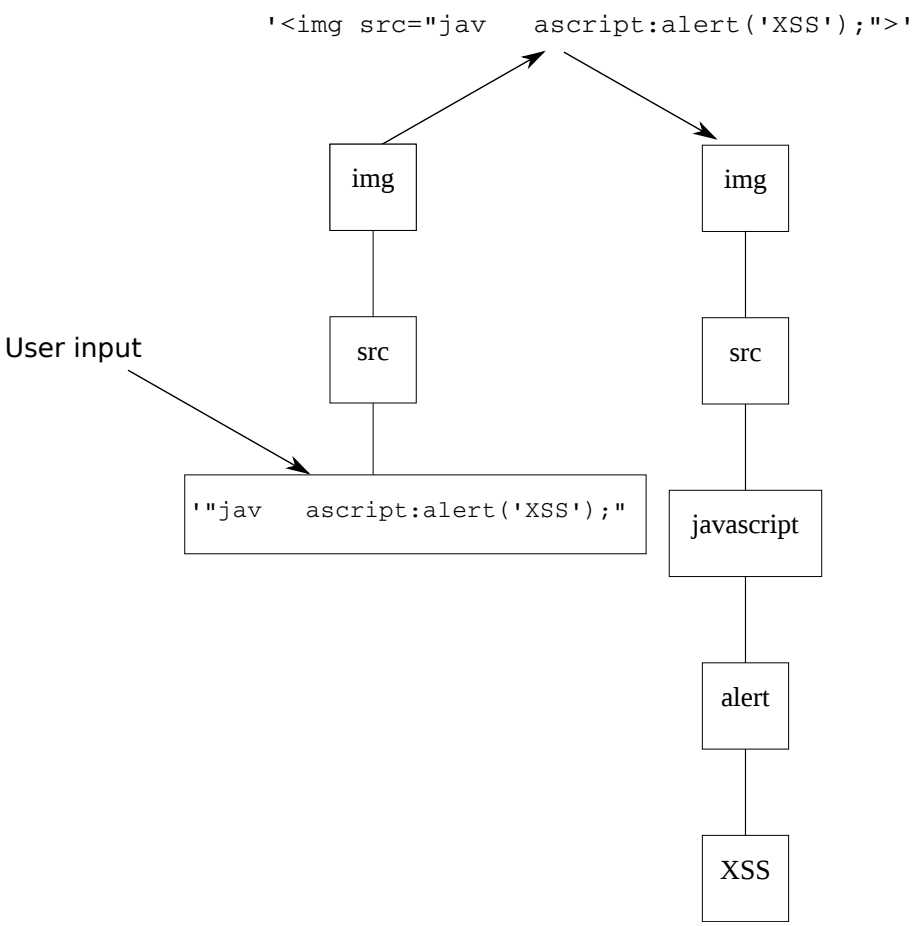

Fig. 2. Concrete example of an XSS style injection attack when working with structures. The AST on the left is proper although it contains a suspicious user input. Naively serializing and deserializing this AST results in the one on the left. The suspicious user input has resulted in an injection attack.

Similar vector for injection attacks arises with strict parsers which are used to parse subtly different languages. This has been known to happen with database engines that interpret different dialects of SQL. For example, some database engines perform Unicode homonym conversion, which normalises Unicode characters into closely matching ASCII characters. This may result in an injection attack regardless whether the manipulation of the original expression has been done with care, but without regard to such homonym conversion.

Part of the above problem arises from the fact that user input formats are simply bad as serialization formats. For instance, consider comments. While adding comments makes it easier to write program code, comments are useless in serialization. When used as intended, comments carry no information for the machine, but when used incorrectly, they enable attackers to remove elements from the resultant AST. Combining this with the possibility of adding structure is especially dangerous as it allows the attacker to *exchange* parts of the AST with similar looking parts.

Thus, to use ASTs to block injection attacks we must ensure that their structure is interpreted identically by all parties exchanging data. For this, we adopt general idea from Proof Carrying Code.

3 Proof carrying code

The concept of Proof Carrying Code (PCC) was introduced by Necula and Lee in 1996 [15]. PCC is a software mechanism for providing the host system means for determining the safety properties of remote code. The idea of proof carrying code is simple. The provider of the remote software component must also provide a *safety proof* for the component. The remote system contains a *verifier* that can validate that the safety proof holds for provided component and that it adheres to the *security policy* demanded by the host system. Should an attacker tamper with either the code or the proof, the host system can reject the code. If the code is tampered with, the verifier cannot validate the safety proof, while if the proof itself is modified, it no longer adheres to the safety policy of the host system causing it to be rejected.

Proof carrying code has multiple advantages. It can be used to run efficient but otherwise unsafe languages, such as C, safely on host systems. It also does not need cryptography to ensure safe execution, which avoids many design issues related to cryptographic systems. However, the downsides of PCC are also significant. Firstly, practical creation of programs with proofs require a *certifying compilers* that are able to provide the proofs [16]. Such compilers are not available for many languages and their construction is a significant undertaking. Also, some desirable properties, such as non-termination can require the manual construction of proofs when the software is constructed. Furthermore, implementing the verifiers can be an advanced undertaking, requiring deep knowledge of first order logics and proof theory.

PCC has been applied to tasks such as kernel mode packet filters [15], transmitting type safety information [15] along with compiled ML programs, verifying

bytecode on smart cards [8]. In our approach, we adopt the PCC idea of proof and verifier to ensure structural integrity of expressions. Unlike the seminal work, our "'proof"' is cryptographic instead of logical.

4 Our approach

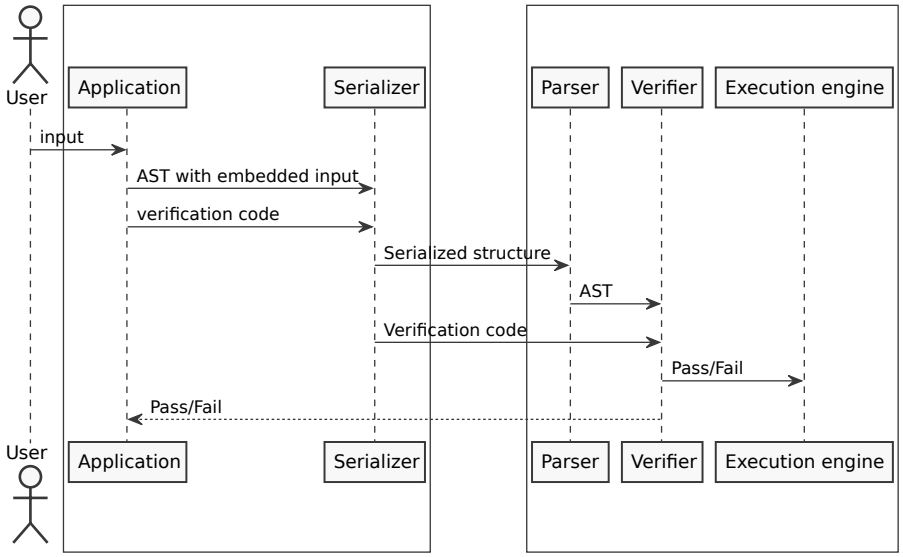

Fig. 3. Overview of our approach to injection protection.

Our procedure for eliminating injection attacks is depicted in Figure 3 and it functions in the following way: Firstly, any system that embeds untrusted input into an expression must do so using structural manipulation of the data. That is, the system may not include uncontrolled structure from user input to enter the AST nor allow the textual representation of the expression to be manipulated in any way. Further, when the user input is embedded into the expression, it must be added as a specific node in the AST.

Then, when serializing the expression for transportation, the system calculates a cryptographic hash according to the (non-serialized) structure of the expression. This hash is then considered as a proof that the document is structurally unmodified. Finally, the resulting structure is serialized in the usual fashion and delivered to other parties along with the structural hash. The deserialization process then parses the serialized format into a structure, verifies that the hash is preserved and thus ensures that no structural changes have taken place. If any changes are present the system flags the input as an injection attack and terminates.

In terms of PCC, our security policy simply states that the structure of the expression may not be interpreted in a different way from what is intended by the application. The cryptographic hash forms the safety proof for this policy, while the verifier is implemented on top of the pre-existing parser using a simple a hash comparison.

Next, we present a concrete algorithm for our proposal. In the following, we assume that means for generating the AST of the expression language is available. Further, we assume that the AST can be generalized into a labeled tree, where the labels represent the operators, variables and constants of the expression language. Specifically, the labels must atomic in the sense that they have no internal structure. For example, HTML attributes which are often represented as strings must be parsed into ASTs and included to the main AST before hashes are calculated.

We also assume that there is a pre-existing serialization procedure for the AST, such as encoding into HTML. We call this procedure "serialize". We then add structural integrity "proof" to this serialization procedure, by defining a serialize' function as

$$\text{serialize'}(a) = H(\text{seq}(a))\|\text{serialize}(a)$$

where the function $H : \mathbb{N}^* \to \mathbb{N}$ is a suitable cryptographic hash function such as SHA-2, The function $\text{seq} : \text{AST} \to \mathbb{N}^*$ is used to generate an encoding for generic tree structures:

$$\text{seq}\,(a) = \begin{cases} 0\|H(L(a)) & \text{when } a \text{ is a variable} \\ 0\|H(L(a)) & \text{when } a \text{ is a constant} \\ n\|H(L(a))\|\text{seq}(a_1)\|\text{seq}(a_2)\|\ldots\|\text{seq}(a_n) & \text{when } L(a) \text{ is has arity } n > 0 \end{cases}$$

here, the function L denotes extracting the label from the AST node.

Finally, if the expression language has comments or equivalent constructions, they must either be included as AST nodes or stripped out completely before the data is transferred between parties.

4.1 Case study: Preventing XSS through verification codes

To gain practical experience with our technique, we implemented a simple web based chat system using the proposed technique. This system includes gathering user input and including it, verbatim, inside document elements. To further test the limits of the technique, we also allow adding user input to the attributes such style and various event attributes. Purely as a test, we also allow including user input inside a string constants of a JavaScript program embedded in the page.

In a short amount time we could obtain structural manipulation for reasonably large subset of HTML5, which is enough to create the above described chat application. We could obtain safe inclusion of arbitrary user input into basic HTML attributes, including style attributes, inside arbitrary DOM nodes, as well as inside script elements.

However, only a limited support for our technique could be obtained without modifying the browser. That is, our current prototype can only generate and verify basic HTML node structure and attributes that contain:

- atomic clauses such as truth values and numeric literals
- URLs, by parsing them through the DOM
- CSS styles, by parsing them through CSSOM
- CSS class lists, by parsing them through the DOM
- JavaScript, by limiting browser support to Firefox, which exports the Spidermonkey JavaScript engine parser.

Regarding the last limitation, other browsers support libraries for parsing JavaScript but do not ensure one-to-one correspondence between library output and the JavaScript engine output, making injection attacks theoretically possible. Finally, regardless of our efforts some CSS style definitions, such as `transform: rotate(7deg);` remain as non-atomic strings.

4.2 Validation

To perform initial validation for our technique, we sampled a set of (essentially different) XSS attacks from the OWASP XSS filter evasion checklist. Naturally, the majority of the attacks described in the checklist rely on truly lax interpretation of the HTML documents and simply parsing them with the browsers `DOMParser` -API [1] allowed the system to discard a large number of them. The remaining attacks were defended against.

Naturally, a more proper validation would be required and we are planning on exposing our XSS-protection system to wider public with a small monetary reward in order to see how it can be circumvented.

4.3 Possible caveats

Our technique is based on the idea that the verifier is able to observe the actual AST which is interpreted by the browser. Erroneous assumptions here can compromise the system. For instance, accidentally obtaining a partial AST and processing it as if it was complete can allow attacker to sneak in an injection attack. Also, current browsers do not provide an uniform API to access the entire AST, requiring us rely on various "hacks" to obtain a proper view into it.

Also, while our technique guarantees that the structure of the document is preserved, does not stop the application developer from intentionally enabling attacks. For example, if the application programmer explicitly constructs a JavaScript node with a call to `eval` function and embeds an user specified string in it, our system will not offer any protection besides ensuring that the string does not escape outside of the `eval` node . Similarly, allowing user to, e.g., choose a function to be called would allow a "return-to-libc" style attack to be performed. In summary, the proposed technique needs to be complemented with common sense and strict interface which hides problems like these.

4.4 Lessons learned

During this experiment, we found that it is surprisingly easy to cover a large enough subset of web techniques to allow simple applications to be written. However, we also noticed that implementing the technique on top of current browsers is a very error prone task. For instance, the current Document Object Model explicitly holds structured data in node attributes as strings ([4], Section 3.2.3.1). Attributes can include complex datums such as JavaScript code and URLs there is no uniform API with which to access their ASTs. Our the prototype implementation must, for this reason, rely on various "tricks" to obtain information how the browser parses such attributes. For example, in our prototype URLs are parsed by creating (and later discarding) an "`a`"-node with the URL as an attribute and then accessing the parsed URL through the methods in the resulting DOM node.

Also, attaining a complete coverage is probably unfeasible without explicit browser support. For example, ordinary `script`-nodes contain nearly arbitrary textual content, including multiple different programming languages. Luckily, our technique allows both the structural editing and verifier to operate on a subset of the actual language. For example, when encountering an unexpected node, our verifier can stop immediately: no unknown node can be generated by the application and therefore it must be the result from an attempted attack.

5 Related work

There are many different targets for injection attacks. For large part, any time when a document is intended for both human and machine consumption, including user input into it may cause injection vulnerabilities. The two most known and contemporarily most damaging injection targets are SQL-databases and HTML pages.

SQL injections pose a great danger to public facing services. An injection attack may lead to unrestricted access to the entire database the service runs on and all the sensitive data therein. SQL injection prevention is widely studied. For example, Halfond et. al. [10] survey several different SQL injection attacks and defenses, claiming that there are many different types of attacks and that many practitioners are aware of only few of them.

Understanding injection attacks as operations that cause improper structural changes to interpreted expressions is common. For example, Su et. al. [17] provide a formal SQL injection protection scheme based on parsing the data. Su et. al inject delimiters around user input. Then the user input is parsed and compared to accepted syntactic forms. However, this approach, as the authors declare "is vulnerable to an exhaustive search of the character strings used as delimiters." This limitation arises from the authors desire for generality and keeping the applications business logic separate from the protection scheme. Furthermore, although the proposed method is immensely effective locally, does not protect across the serialization/deserialization.

There are two approaches for SQL injection protection that we know that apply similar strategy as we propose. Firstly, the SQL DOM proposed by McClure and Krüger [13] is a way of generating and abstract object representation of given database schema. This corresponds to the structural manipulation part of our proposed technique.

Secondly, there is a well known approach called SQLGuard introduced by Buehrer et. al., which applies an SQL parser to queries where user input is injected. Should the parsed structure differ from the intended structure, or the parse tree of the template query, the system flags the input as an injection. We endorse this technique for similar reasons as SQL DOM. Unlike the more principled SQL DOM, SQLGuard can be retrofitted to existing applications easily and it effectively provides the same safeguards. On the other hand, dynamically structured queries are not possible using SQLGuard, whereas SQL DOM has no difficulties with them.

Cross site scripting (XSS) is another common and extremely persistent form of an injection attack. XSS vulnerabilities are caused by inadvertently allowing the inclusion of attackers JavaScript code on a HTML page. Such scripts have the same capabilities as the user of the page, allowing the attacker access to users private data.

The literature of XSS protection schemes is systematically reviewed by Hydara at. al. in 2015 [12]. The authors conclude that at the time of writing there exists significant number of studies on defending against XSS. The protection schemes are varied and range from static analysis [3] to meta heuristic approaches [2]. The authors however conclude that there is currently no single solution to effectively mitigate all XSS attacks, and call for more research on the matter.

The idea of structural preservation is not new in the context of XSS. For example, Barhoom et. al. [7] propose to overlay the input portions of the web page with generated XSD schemas [9] which are then used to validate the user input by comparing its *structure* to the one stored in the schema. Our approach has similar features to the scheme proposed Barhoom et. al.

An another example of doing structural preservation appears in work by Nadji et. al. [14]. Here, the authors propose enclosing the user input with special delimiters so the browser is able to adhere to this demarcation when parsing the web page with the input. Naturally, the authors note that simply adding a new set of delimiters only adds another site for injections. The authors instead propose using a random set of delimiters, information of which is passed from server to browser through some secure channel. This solution works on the same basic principle as ours, but it is much more complex. However, it may have a greater potential in retrofitting large amounts of broken server code than ours.

Ter et. al. [18] also consider the preservation of document structure as a key point in XSS prevention. Like us, Ter et. al. note the differences between browser parsers and possible server side parsers as an enabler for XSS attacks and propose a radical solution: doing entirely away with the said parsers when handling user input. Surprisingly, this technique is compatible with current browsers with only

slight added latency. Our technique could be also adapted for current browsers using similar technique.

6 Discussion

In this article we propose a systematic way to eliminate classical injection attacks entirely by enabling applications work directly with ASTs while ensuring uniform interpretation of the result. Our technique requires a complete move away from processing data as strings.

We managed to implement a non-trivial XSS-protected web application with our technique in a limited time. We think that the implementation cost of the technique on a green field application is reasonable. Retrofitting pre-existing software with this technique could, however, be seen as difficult task. However, we claim that moving away from manipulating textual representations of data is the "right thing to do". Injection attacks are not the only problem that arises in "stringly" typed programs. It is not difficult to conceive operations that generate badly formed expressions during runtime, causing hard to detect bugs in already delivered software components.

On retrofitting old software, the first steps would be to identify all procedures that can be used to send out data and replace these with versions that accept only structured data. Identifying all call sites of these procedures can be done with standard static analysis techniques, or even bluntly commenting out the methods and thus identifying the call sites from compiler error messages.

Regarding other lessons learned, we importantly noted that many, if not most, libraries that allow structured HTML generation handle attribute values as strings. In our view, this increases the likelihood of successful injection attacks by encouraging programmers to pass values as is from user to the page. We advocate that any future library for manipulating HTML should work with non-opaque representations of HTML attributes.

7 Acknowledgements

I would like to thank Maria Tirronen for her time and help in editing this article.

References

1. Dom parsing and serialization. W3c working draft, W3C (May 2016), https://www.w3.org/TR/DOM-Parsing/
2. Adi, E.: A design of a proxy inspired from human immune system to detect sql injection and cross-site scripting. Procedia Engineering 50, 19–28 (2012)
3. Agosta, G., Barenghi, A., Parata, A., Pelosi, G.: Automated security analysis of dynamic web applications through symbolic code execution. In: Information Technology: New Generations (ITNG), 2012 Ninth International Conference on. pp. 189–194. IEEE (2012)

4. Arnaud, L.H., Philippe, L.H., Lauren, W., Gavin, N., Jonathan, R., Mike, C., Steve, B.: W3c recommendation, W3C (May 2004), https://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/
5. Athanasopoulos, E., Pappas, V., Krithinakis, A., Ligouras, S., Markatos, E.P., Karagiannis, T.: xjs: practical xss prevention for web application development. In: Proceedings of the 2010 USENIX conference on Web application development. pp. 13–13. USENIX Association (2010)
6. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: Composing static and dynamic analysis to validate sanitization in web applications. In: 2008 IEEE Symposium on Security and Privacy (sp 2008). pp. 387–401. IEEE (2008)
7. Barhoom, T.S., Kohail, S.N.: A new server-side solution for detecting cross site scripting attack. Int. J. Comput. Inf. Syst 3(2), 19–23 (2011)
8. Cho, J.B., Jung, M.S.: A very small bytecode-verifier based on pcc algorithm for smart card. In: International Conference Human Society@ Internet. pp. 106–115. Springer (2003)
9. Gao, S., Sperberg-McQueen, C.M., Thompson, H.S., Mendelsohn, N., Beech, D., Maloney, M.: W3c xml schema definition language (xsd) 1.1 part 1: Structures. W3C Candidate Recommendation 30(7.2) (2009)
10. Halfond, W.G., Viegas, J., Orso, A.: A classification of sql-injection attacks and countermeasures. In: Proceedings of the IEEE International Symposium on Secure Software Engineering. vol. 1, pp. 13–15. IEEE (2006)
11. Heiderich, M., Schwenk, J., Frosch, T., Magazinius, J., Yang, E.Z.: mxss attacks: Attacking well-secured web-applications by using innerhtml mutations. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 777–788. ACM (2013)
12. Hydara, I., Sultan, A.B.M., Zulzalil, H., Admodisastro, N.: Current state of research on cross-site scripting (xss)–a systematic literature review. Information and Software Technology 58, 170–186 (2015)
13. McClure, R.A., Kruger, I.H.: Sql dom: compile time checking of dynamic sql statements. In: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005. pp. 88–96. IEEE (2005)
14. Nadji, Y., Saxena, P., Song, D.: Document structure integrity: A robust basis for cross-site scripting defense. In: NDSS. vol. 2009, p. 20 (2009)
15. Necula, G.C.: Proof-carrying code. design and implementation. In: Proof and system-reliability, pp. 261–288. Springer (2002)
16. Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. ACM SIGPLAN Notices 33(5), 333–344 (1998)
17. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: ACM SIGPLAN Notices. vol. 41, pp. 372–382. ACM (2006)
18. Ter Louw, M., Venkatakrishnan, V.: Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In: 2009 30th IEEE Symposium on Security and Privacy. pp. 331–346. IEEE (2009)