Preventing Cross-Site Scripting with Script-Free HTML

Matthew Seffernick

ABSTRACT The injection of scripts into a web page by means of evading input filtering is called a cross-site scripting (XSS) attack. Even popular websites, such as Google, Facebook, and YouTube, have been exploited by XSS attacks. In 2010, OWASP ranked XSS attacks the 2nd-leading source of web security risk. Current methods to prevent XSS exploits are either ineffective (allowing some attacks to succeed) or overly prohibitive (preventing legitimate HTML-rich content). This paper describes a new approach: the structure of safe input is rigorously defined and a server-side tool is implemented to detect the presence of a potential XSS attack. This tool prevents XSS attacks while still permitting HTML-rich content. We define a new context-free grammar (Script-Free HTML 4) that precisely characterizes safe input. Our approach is evaluated by applying it to a benchmark of known XSS vulnerabilities. We also consider the future evolution of this approach in the ever-changing world of web standards.

INTRODUCTION

Background

Web pages are often generated dynamically by combining static templates with dynamic content. Because the templates are static and changed only by the website staff, website administrators can reason that the templates produce web pages with desired appearances and behavior. However, website administrators may not fully know how the behavior and appearance of web pages produced by their templates may be affected by incorporating dynamic content. In some cases, the resulting web page also produces desired appearance and behavior. In other cases, however, it may cause visitors' web browsers to execute code the web administrators consider undesirable, such as sending the visitor's cookie data to another website. Thus, we consider any dynamic content that can be manipulated by user input to be untrusted. We call user input which creates or attempts to create malicious dynamic content an attack vector.

Ligatti and Ray define code-injection attacks as attacks in which untrusted content is used as code (Ray & Ligatti, 2012). Shar and Tan define XSS attacks as attacks in which web browsers treat untrusted content as scripting content (Shar & Tan, 2012). As these definitions demonstrate, the identification and prevention of XSS attacks requires two things: an understanding of when untrusted content may be interpreted as browser scripting content and how attackers bypass preventative measures to inject scripting content into web pages served to others.

According to the HTML 4.01 Strict Doctype Definition (DTD), web browsers should only invoke scripts within script elements and event attributes. As highlighted by Bisht and Venkatakrishnan, however, browser quirks introduce major difficulty and uncertainty in identifying potential scripting content (Bisht & Venkatakrishnan, 2008). For example, some versions of Internet Explorer invoke JavaScript when "javascript:" starts the IMG element's src attribute value. Additionally, embedded content, such as Cascading Style Sheets (CSS) and Flash files, can contain browser-side scripting content and is impossible to detect without also scanning the embedded content (OWASP, 2013). Thus, to prevent scripting invocations based on only a web page's HTML, one must prevent untrusted content from: deviating from well-defined HTML; abusing known web browser quirks; and affecting or introducing script elements, event attributes, and embedded content.

As demonstrated by OWASP's filter evasion cheat sheet samples (made by RSnake), a successful attack vector produce untrusted content that interacts with trusted content by opening or closing elements and starting or ending attributes (OWASP, 2013). When untrusted content is not restrained, an attack vector can trivially invoke JavaScript; to end an attribute, untrusted content needs only to contain a quotation mark; and to close an element, untrusted content needs only to contain the element's closing tag. As a result, unchecked input leaves a site completely vulnerable to XSS attacks.

Restraining untrusted content restricts the ways in which user input may be used and increases the burden of creating feature-rich web applications, however. For instance, the safest use of untrusted content is to force every character to be printed, not interpreted, by the web browser (such as when a search engine displays the search query) or to check the input for an enumerated value and to behave in response in a well-defined way (such as using a checkbox to determine if or how dates should be displayed). The tradeoff may be acceptable to banking and search websites, where the focus is on the service provided. Websites whose focus is on user content, such as forums or blogs, however, may prefer to permit users direct access to some HTML features for formatting text and structuring web pages.

Related Work

SQL injection attacks are a web application security problem similar to XSS attacks. In an SQL injection attack, an attacker sends an attack vector to a web application with the intent of the web application executing the string as code in an SQL statement rather than as a literal value (Su & Wassermann, 2006). As a result of SQL injection attacks, attackers may steal information from the website (such as obtaining account details and login credentials) or remove information from the site (such as dropping a database).

An approach to detect these attacks at run-time is to create and compare SQL query parse trees (Buehrer, Weide, & Sivilotti, 2005). A parse tree represents the structure of an SQL statement, where leaves of the tree represent specific tokens (keywords, identifiers, and literals) and nodes of the tree represent groups of tokens. Assuming web applications always intend for untrusted content to be a proper subtree in an SQL parse tree, the parse tree generated as the result of an SQL injection attack necessarily will not syntactically match that of the parse tree generated by the SQL query intended by the web application.

To apply this approach to XSS attacks, we would compare JavaScript parse trees rather than SQL parse trees. However, web pages sent to visitors consists of HTML, which web browsers interpret to invoke JavaScript. Thus, we would need to work with HTML parse trees rather than JavaScript parse trees. Then we would need to determine what subtrees of an HTML parse tree invoke JavaScript and detect when subtrees constructed from untrusted content contain subtrees that invoke JavaScript. The application would detect potential XSS attacks but would not respond to them.

An approach to detect SQL injection vulnerabilities before run-time is to first describe, via a grammar, all possible SQL queries generated by untrusted content and application code; then determine if the resulting grammar will permit an SQL injection attack (Wassermann & Su, 2007). In their implementation, Wassermann and Su use context-free grammar rules to describe the changes a string may undergo by PHP operations and methods. Once the grammar for the given code is generated, they test if the grammar can produce a syntactically open SQL statement, such as if a query string can contain an odd number of non-escaped quotation marks.

To apply this approach to XSS attacks, we would define our target language to be HTML but excluding all JavaScript invocations. We would then analyze the web application code, produce a grammar for all possible web pages, and determine if the produced grammar would permit a syntactically open HTML page. Web applications would then scan their applications before exposing the application to live web traffic, fix any detected issues, and rescan after fixing.

Summary of Existing Approaches

Shar and Tan highlight 3 simple, yet popular and effective, methods for preventing XSS attacks: blacklisting, whitelisting, and character escaping. Blacklists describe unsafe input that should be rejected when encountered. Typically, web applications blacklist attack vectors by describing them with regular expressions and scanning attack vectors and untrusted content for matches. When the application encounters a match, the application removes or replaces the matched substring or rejects the entire string. Whitelists, on the other hand, describe safe input; web applications reject input that does not match values matched by its whitelist. Character escaping replaces all HTML meta-characters with their HTML-encoded equivalents. That is, any character in user input which would normally have special meaning to a web browser is replaced with text which instructs web browsers to display the character rather than interpret it (Shar & Tan, 2012).

XSS-Guard generates a webpage twice, once with user input and once with safe input. Both webpages send their input through the same path through the web application's code: the path the user input takes. Using a parser derived from FireFox's content sink, XSS-Guard generates a JavaScript parse tree for each page and compares the parse trees. If XSS-Guard finds that the parse trees are syntactically equivalent, XSS-Guard considers the generated output safe and does nothing. Otherwise, it alters the scripting content sent to the user, replacing what it identifies as malicious scripting content with a benign replacement. Thus, XSS-Guard uses the generated scripting content to determine safeness rather than just the untrusted content (Bisht & Venkatakrishnan, 2008).

HTML Purifier uses the structure of HTML and a whitelist of HTML elements and attributes to maintain a description of safe input. The application parses input into tokens, alters and validates input according to its settings, and converts the resulting tokens back into a string for use by the web application. HTML Purifier intends for web applications to receive user input, pass the user input through HTML Purifier, and embed the returned text straight into the output page (Yang, 2012).

Analysis of Existing Approaches

Due to their simplicity, blacklists, whitelists, and character escaping are efficient and easy to use. However, as Shar and Tan indicate, they each have issues. Blacklists tend to fail to catch all attack vectors; whitelists prohibit much safe, valid content; and character escaping prevents use of HTML-rich content (Shar & Tan, 2012). As a result, web developers cannot use these methods to permit HTML-rich content, such as an entire self-structured web page on a blog site or self-formatted text in a forum post.

XSS-Guard more effectively detects scripting content due to its high-level approach, but at the cost of generating and comparing two JavaScript parse trees and sanitizing scripting output. Additionally, XSS-Guard fails when web applications use conditional copying, such as when one string is copied character by character to another string which is then included in the output HTML response. XSS-Guard also fails when an exploit is embedded in a Flash object included by the web application (Bisht & Venkatakrishnan, 2008).

HTML Purifier's enforcement of a strict HTML structure and an element and attribute whitelist effectively prevents most risky user input. However, HTML Purifier's behavior is difficult to reason about due to its highly configurable whitelist and lack of clearly defined behavior for detecting and editing. Additionally, HTML Purifier does not sanitize input based on the context in which the input will be used, potentially leading to unsafe use of "safe" input, an issue discussed by Ligatti and Ray.

Objective

In this paper, we present a new approach for preventing XSS attacks. We describe safe content, HTML free of JavaScript invocations, with a context-free grammar and implement a parser for the grammar. Web applications can then use the parser to verify that untrusted content will not produce JavaScript invocations.

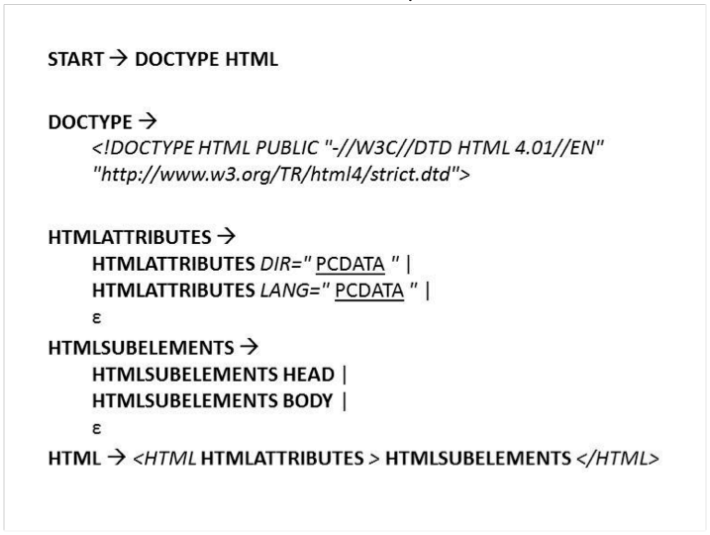
METHODOLOGY

Summary

To address the problem of XSS attacks, we develop: a context-free grammar, called Script-Free HTML 4 (SFH4), which produces a language that follows the structure of HTML and is free from JavaScript invocations; a parser for SFH4; and a methodology for generating a context-free grammar and parser from a Document-Type Definition (DTD).

SFH4 determines if input is safe or unsafe on the basis of the input's HTML structure and scripting content. For instance, if untrusted content consists of well-formed HTML 4 with no signs of possible script invocations, SFH4 accepts it as safe. Conversely, if untrusted content contains malformed HTML 4 or content which follows a pattern known to risk browser script invocation, SFH4 rejects it as potentially unsafe. Thus, SFH4 is sound but imprecise; it only accepts safe input, but also rejects input which may, in practice, be safe. For a snippet of the grammar, see Figure 1.

SFH4's rewrite rules follow the structure of an XML document but contain only the subset of elements



```
START → DOCTYPE HTML

DOCTYPE →
    <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">

HTMLATTRIBUTES →
    HTMLATTRIBUTES DIR=" PCDATA " |
    HTMLATTRIBUTES LANG=" PCDATA " |
    ε
HTMLSUBELEMENTS →
    HTMLSUBELEMENTS HEAD |
    HTMLSUBELEMENTS BODY |
    ε
HTML → <HTML HTMLATTRIBUTES > HTMLSUBELEMENTS </HTML>
```

Fig.1. In the given grammar snippet, tokens are nonterminals are in bold, terminals are in italics, and tokens are underlined.

and attributes from the HTML 4.01 Strict DTD we deemed safe. To decide which elements and attributes were safe, we reviewed each element and attribute from the HTML 4.01 Strict DTD and the attack vector samples made available by OWASP (OWASP, 2013). The contents of the whitelist can be altered without affecting our approach. Of the available elements, we permitted all but base, link, meta, object, param, script, and style. Most of these elements were not permitted due to risk from embedding content. Of the available attributes, we permitted all but onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup, action, profile, onfocus, onblur, and style. Most of these attributes were not permitted due to explicit JavaScript invocations. Due to a known browser quirk with some versions of Internet Explorer, we also defined a special token for the img element's src attribute to permit only URLs.

To use SFH4, a web application must run a server-side parser for the grammar and construct test pages to pass to the parser. To see the context in which an SHF4 parser would be used, see Figure 2. Test pages contain the untrusted content the web application wishes to verify as well as surrounding HTML to define the context in which the untrusted content will be used. The test page also separates the trusted content, which may contain JavaScript invocations, from the untrusted content, which should not. For a sample test page, see

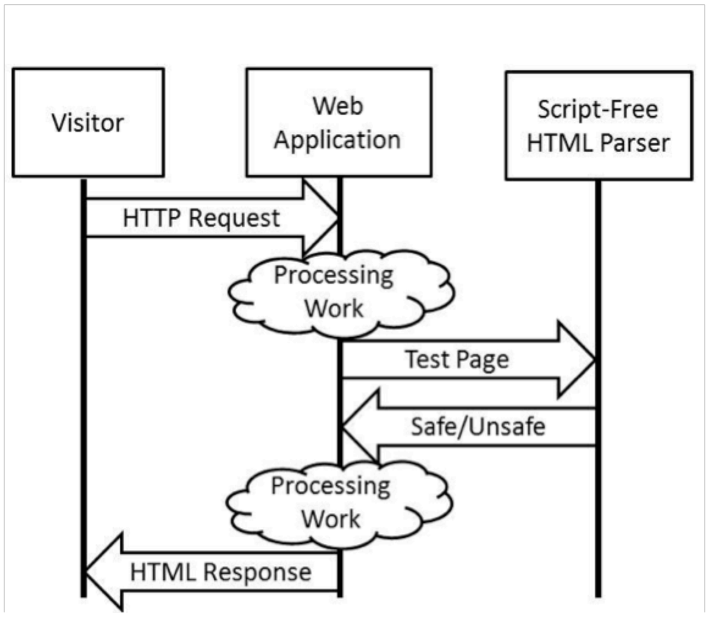


Fig. 2. SFH scans untrusted content on the server after the server receives a request but before the server sends a response.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML
4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html><body><div>
Untrusted content
</div></body></html>
```

Fig. 3. To scan untrusted content, web applications must provide SFH with the context in which the untrusted content will be used.

Figure 3.

If the parser accepts the test page, the untrusted content follows HTML structure and contains no elements or attributes at risk for invoking JavaScript. This judgment may vary based on the HTML surrounding the untrusted content, however, so untrusted content must be rescanned with a new test page for each context in which it is used. If the parser rejects a test page, the web application may sanitize the untrusted content and rescan it, or the web application may reject the untrusted content and request new input; the web application should not use the untrusted content as-is in the context intended.

Implementation

In generating and implementing our SFH4 grammar, we used the HTML 4.01 Strict DTD as input, mIRC Scripting Language (MSL) from mIRC v7.17 for generating token and grammar definitions, GNU Bison v2.4.1 and GNU Flex v2.5.4 to generate the C code for the scanner and parser, and Gnu C Compiler (gcc) v4.6.1 to compile the scanner and parser code. To summarize the process before going into detail, we take a Document Type Definition (DTD) and manually created configuration files as input and, in steps 1 and 2, parse the DTD to create a whitelist of permitted elements and attributes. In step 3, we alter the whitelist of elements and attributes to exclude the elements and attributes we deem unsafe. In step 4, we generate the grammar using the whitelist of elements and attributes and configuration files. In the remaining steps, the Bison and Flex applications generate C code for the parser using our grammar definition, and gcc compiles the C code into the final executable. For an overview of the workflow, see Figure 4.

Firstly, an MSL procedure (parseItems) parses the DTD for element, attribute, and entity definitions. Secondly, more MSL procedures (evaluateEntities, evaluateElements, and evaluateAttributes) evaluate the definitions and parse them into two lists: a list of ele-

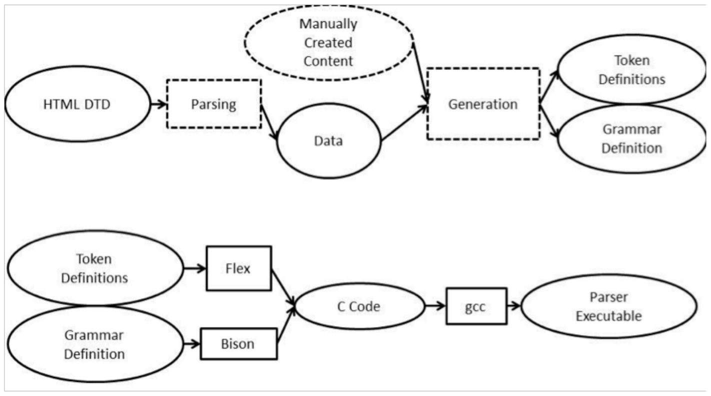


Fig.4. Our contribution in the shown workflow diagram has a dotted border. Input and output are rounded shapes, and processes are rectangular shapes.

ments and their permitted sub elements, and a list of elements and their attribute declarations. In the third step, another MSL procedure (removeItems) modifies the two lists to remove unsafe elements and attributes and restrict the values of attributes with known browser quirks. Rather than removing elements and attributes from the parsed data in step 3, we could have removed element and attribute definitions from the DTD before passing the DTD to the parsing process.

With exceptions from browser quirks, attack vectors invoke JavaScript via unsafe attributes, not particular values within otherwise safe attributes. As such, we default the value type of all attributes to PCDATA and make special exceptions for known browser quirks. By defaulting attribute value types to PCDATA, we lower the number of tokens we must manually define.

In the fourth step, another MSL procedure (makeInputFiles) reads the parsed data and configuration files to generate the token and grammar definitions. To generate the token definitions, the procedure reads from a configuration file containing definitions for token patterns and code to execute when a pattern is matched. With the code written in C, we use global variables to track whether the current token is inside a tag, inside an attribute, or in-between tags. We give each element and attribute ID its own token and use MSL to procedurally generate a method (parseID) which, given an input string, returns the token for the ID if it is on the whitelist or an error token otherwise, thus prohibiting blacklisted and unknown element and attribute IDs.

To generate the grammar definition, makeInputFiles reads two configuration files. The first file contains: a C prototype for a method (yylex) required by Bison and implemented by Flex, specially defined scanner tokens (PCDATA, URI, ERROR, and DOCTYPE), and a start rule for the grammar. The second configuration file contains implementations of two C methods (yyerror and main) required by Bison. makeInputFiles procedurally generates scanner tokens for every element and attribute ID, as well as the grammar rewrite rules for all elements and attributes.

In the fifth step, the token and grammar definitions are passed to Flex and Bison, generating C code for the scanner and parser. Finally, gcc compiles the C code with gcc to create the parser executable.

RESULTS

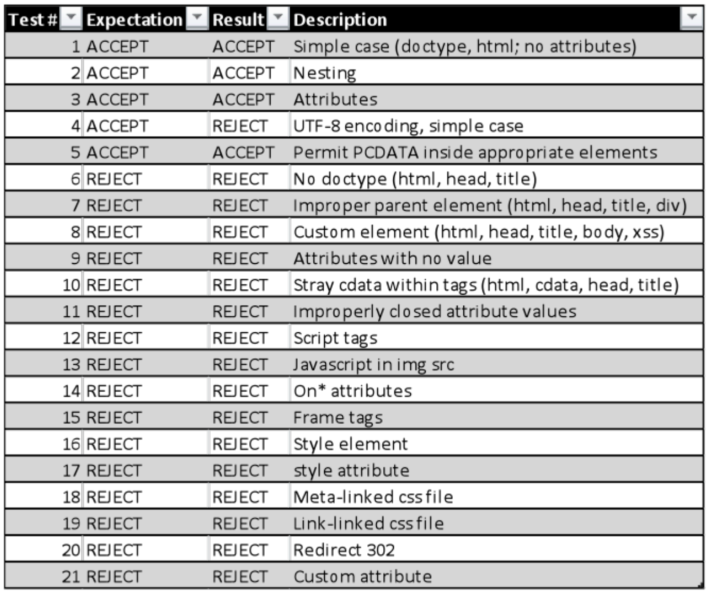
To test our approach, we implemented and compiled our SFH4 parser and passed it input from files. To test the soundness of our approach, we passed the parser 16 attack vectors based on OWASP's filter evasion cheat sheet. To test the precision of our approach, we passed the parser 5 examples of safe input based on the HTML 4.01 Strict DTD. Finally, to test the performance of our implementation, we passed each of our test cases, as well as an additional 2.8KB test case to simulate a large test page, to the parser 20 times. For descriptions and results of test cases used, see Table 1.

Our implementation rejected all 16 attack vectors it should have rejected, but it only accepted 4 of the 5 cases of safe input it should have accepted. In the rejected safe case, the parser rejected UTF-8 encoded text, whereas the other test cases used ASCII-encoded text. The system used to test performance of our implementation had an 8-core Intel(R) Xeon(TM) CPU at 2.80GHz per core, and 8059128 kB of memory. The tests ran with a mean of 0.022 seconds per set of 22 tests, or 0.001 seconds per test, with a variance of 0.004 seconds.

DISCUSSION

Key Findings

Upon analyzing common attack vectors, we found that most attack vectors could be generalized to a few patterns. In general, attack vectors: violate HTML structure, embed foreign content, invoke scripts in CSS, or exploit browser quirks. Enforcing a strict HTML structure handles the case of input violating HTML structure, and enforcing an element and attribute whitelist handles the cases of input embedding foreign content and CSS. Because browser quirks vary by version and



Test #	Expectation	Result	Description
1	ACCEPT	ACCEPT	Simple case (doctype, html; no attributes)
2	ACCEPT	ACCEPT	Nesting
3	ACCEPT	ACCEPT	Attributes
4	ACCEPT	REJECT	UTF-8 encoding, simple case
5	ACCEPT	ACCEPT	Permit PCDATA inside appropriate elements
6	REJECT	REJECT	No doctype (html, head, title)
7	REJECT	REJECT	Improper parent element (html, head, title, div)
8	REJECT	REJECT	Custom element (html, head, title, body, xss)
9	REJECT	REJECT	Attributes with no value
10	REJECT	REJECT	Stray cdata within tags (html, cdata, head, title)
11	REJECT	REJECT	Improperly closed attribute values
12	REJECT	REJECT	Script tags
13	REJECT	REJECT	Javascript in img src
14	REJECT	REJECT	On* attributes
15	REJECT	REJECT	Frame tags
16	REJECT	REJECT	Style element
17	REJECT	REJECT	style attribute
18	REJECT	REJECT	Meta-linked css file
19	REJECT	REJECT	Link-linked css file
20	REJECT	REJECT	Redirect 302
21	REJECT	REJECT	Custom attribute

Table 1. Our parser passed all test cases except an encoding issue, which is implementation-specific

brand of web browser, browser quirks are difficult to handle and impossible to predict. Thus, an important step to preventing XSS attacks is for web browsers to only invoke scripts in standardized cases.

Our approach requires user input be scanned for every context in which it is used. Input that is syntactically correct and safe in one context (such as between HTML div tags) may not be syntactically correct or safe in another context (such as inside the open tag of an HTML div element). Thus, validating input is not as simple as scanning it once before storing it or scanning it and then using it in multiple locations. The only safe solutions are to validate input before each of its uses or to track the contexts in which the input will be used and validate the input once for each context before use.

Our SFH4 parser successfully rejected all the categories of attacks presented on OWASP's filter evasion cheat sheet, demonstrating our approach's ability to detect potential XSS attacks. The test case in which our parser failed, parsing UTF-8 encoded text, shows a limitation on our implementation but not our approach. With a mean runtime of 0.001 seconds per test on hardware used as a webserver, the runtime is very acceptable for real world application.

Future Work

One shortcoming of our implementation is that it can only handle ASCII-encoded text, thus preventing it from working with non-english characters. To correct this, our parser will need extended to handle other character encodings, such as UTF-8. Another shortcoming of our implementation is that it launches a new process for each test page it parses, thus introducing processing overhead that may not scale to heavily-visited websites. To address this, our parser will need altered to run as a service and accept multiple calls before exiting. Finally, our approach currently gives web applications no information other than whether or not a test page matches our description of safe input. Thus, our approach can be extended to correct unsafe input.

We tailored our approach around HTML 4 because of its widespread use and defined standards, but as web standards change, our approach needs to keep up to remain useful. To support HTML 5, the newest HTML standard, we would need only to write a DTD for safe HTML 5. Any DTD will work with our existing work pipeline to produce an HTML parser, but HTML 5 does not have a DTD, so a DTD will need to be written for it. To support CSS 3, the newest CSS standard, our work pipeline will need altered to accept CSS's structure. Otherwise, we can still use a context-free grammar to describe safe CSS, content which our approach currently prohibits.

Our implementation expects web applications to construct test web pages mimicking the context in which the user input will be embedded. Thus, our implementation is not particularly simple to use on its own and may be prone to human error from the web developer. A solution to this problem may be to package the parser executable with development libraries that interface with the executable. For instance, web developers might import a PHP library which: automatically generates the prefix and suffix content for the test page; runs the test page through the parser, encapsulating the call to exec; and returns a value for "safe" and a value for "unsafe" depending on the parser's exit status. Such a function should make the parser more convenient to use and facilitate tracking of contexts in which input has been validated.

REFERENCES

Bisht, P., & Venkatakrishnan, V. N. (2008). XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment, 23-43. Paris: Springer-Verlag. doi:10.1007/978-3-540-70542-0_2

Buehrer, G., Weide, B. W., & Sivilotti, P. A. (2005). Using parse tree validation to prevent SQL injection attacks. In Proceedings of the 5th international workshop on Software engineering and middleware (pp. 106-113). New York: ACM. doi:10.1145/1108473.1108496

OWASP. (2010). Category:OWASP Top Ten Project. Retrieved February 7, 2013, from OWASP: The Open Web Application Security Project: https://www.owasp.org/index.php/Top_10

OWASP. (2012, November 25). Retrieved February 1, 2013, from OWASP: https://www.owasp.org/index.php/Main_Page

OWASP. (2013, January 25). XSS Filter Evasion Cheat Sheet. Retrieved February 1, 2013, from OWASP: https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Ray, D., & Ligatti, J. (2012). Defining code-injection attacks. In Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 179-190). New York: ACM. doi:10.1145/2103656.2103678

Shar, L. K., & Tan, H. B. (2012). Defending against cross-site scripting attacks. Computer, 55-62. doi:10.1109/MC.2011.261

Su, Z., & Wassermann, G. (2006). The essence of command injection attacks in web applications. In Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 372-382). New York: ACM. doi:10.1145/1111037.1111070

Wassermann, G., & Su, Z. (2007). Sound and precise analysis of web applications for injection vulnerabilities. In Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (pp. 32-41). New York: ACM. doi:10.1145/1250734.1250739

Yang, E. (2012). Retrieved February 1, 2013, from HTML Purifier: http://htmlpurifier.org/