XSS-Dec: A Hybrid Solution to Mitigate Cross-Site Scripting Attacks

Smitha Sundareswaran, Anna Cinzia Squicciarini

XSS-Dec: a Hybrid Solution to Mitigate Cross-Site Scripting Attacks

Smitha Sundareswaran, Anna Cinzia Squicciarini

College of Information Sciences and Technology
The Pennsylvania State University
{sus263,acs20}@psu.edu

Abstract. Cross-site scripting attacks represent one of the major security threats in today's Web applications. Current approaches to mitigate cross-site scripting vulnerabilities rely on either server-based or client-based defense mechanisms. Although effective for many attacks, server-side protection mechanisms may leave the client vulnerable if the server is not well patched. On the other hand, client-based mechanisms may incur a significant overhead on the client system. In this work, we present a hybrid client-server solution that combines the benefits of both architectures. Our Proxy-based solution leverages the strengths of both anomaly detection and control flow analysis to provide accurate detection. We demonstrate the feasibility and accuracy of our approach through extended testing using real-world cross-site scripting exploits.

1 Introduction

Some of the most well-known and significant vulnerabilities of a Web application are related to cross-site scripting (XSS) [15]. XSS vulnerabilities enable an attacker to inject malicious code into Web pages from trusted Web servers. Typically, when the client receives the document, it cannot distinguish between the legitimate content provided by the Web application and the malicious payload inserted by the attacker. Since the malicious content is handled as the content from the trusted servers, it has the privileges to access the victim users' private data or take unauthorized actions on the user's behalf.

XSS vulnerabilities have been analyzed by a number of researchers and practitioners. One of the most common defense mechanism currently deployed consists of input validation at the server-end, wherein the untrusted input is processed by a filtering module that looks for scripting commands or meta-characters in untrusted inputs. The filtering module then filters any such content before these inputs get processed by the Web application. However, proper input validation is challenging; XSS attacks can be crafted so as to bypass the input sanitization steps. Further, input validation adds a significant burden on the server end, and leaves clients defenseless in case of unprotected sites. Consequently, recent efforts have shifted their attention on client-end solutions, to protect client systems against servers that failed to filter untrusted content [13]. Unfortunately, neither one of these approaches are able to withstand against all forms of XSS attacks. For example, server-side solutions require good control and knowledge of the server's source code, and therefore fare well with attacks that target servers, or that

a reflected off Web servers [11, 8, 4]. Client-side solutions instead are most effective against attacks that are perpetrated through attacks which malicious code is contained in the client side page [13, 3]. In this paper, we propose a novel approach to combine the benefits of both server-side and client-side defense mechanisms. We leverage the information obtained from both the client and the server-side using a simple, yet effective, security-by-Proxy approach. Our design allows us to uphold users' browsing activities while thoroughly monitoring the sites' vulnerabilities before any attack is carried out. Specifically, the Proxy develops an anomaly-based detection mechanism, enriched with detailed control flow analysis. These two techniques combined together enable early detection of subtle attacks that may involve obfuscation attempts. In addition, control flow analysis helps us validate any redirections and minimizes the leakage of the victim's information through malicious links.

The architecture includes a Plug-in on the client-end. The Plug-in is responsible for ensuring that any Web site visited by the user is checked by the Proxy. Based on the input of the Proxy, the Plug-in also deploys the actual protection. In either case, we effectively stop the attack from being successfully carried out and affect the user's system. The Plug-in is carefully designed so as to maintain limited amount of information of the user's browsing history.

We extensively test our solution over a large number of actual XSS vulnerabilities. Our evaluation results show not only that we are able to protect against all types of XSS attacks, but also that our approach is efficient and does not impose a significant burden either on the client or on the server. In summary, the key benefits of our solution are:

1. **User Friendly**. Our approach does not require any significant level of human involvement. It is based on a simple Plug-in that interacts with the user to inform him of possible attacks and stop them from being carried out.
2. **High Accuracy**. Our approach can detect all known types of XSS script injections, by providing different levels of protection, that include selectively blocking portions of the sites being infected and preventing the site from being accessed.
3. **Acceptable Overheads**. Our approach does not impose any burden on Web application performances. The overhead at the client-side is minimal, most of the computation is carried out by a Proxy. The Proxy is also very efficient, and therefore it can be used to protect multiple users at the same time.

The rest of the paper is organized as follows. Next section provides an overview of XSS attacks. Section 3 provides an overview of our solution, the XSS-Dec. In Section 4 we describe the Proxy's architecture. In Section 5, we discuss the Plug-in. In Section 6 we present our evaluation results. Section 7 analyzes existing body of work in this area, and Section 8 concludes the paper.

2 XSS Attacks and Common Solutions

XSS attacks are a class of code injection attacks caused by the server's lack of input validation and are typically the result of insecure execution of JavaScript, although non-JavaScript vectors, such as Java, ActiveX, or even HTML, may also be used to mount the attack. XSS attacks can be segregated into the following classes:

DOM-based attacks: The attacker sends a specially crafted URL to the victim, altering the DOM structure of the Web page once it's loaded in the browser. The actual source code is not changed. It is often launched using the `document.location` DOM and then used to populate the page with dynamically generated content.
Reflected XSS attacks: The attack code is "reflected" from a Web server. The attacker inserts malicious JavaScript into some form, which typically reflects the string back to the trusted site, using the content inserted to generate a response on the fly. The attack code, which is treated as belonging to the same domain as the rest of the site, is then executed. This is the most common form of attack.
Stored XSS attacks: In this type of attack (also known as HTML injection attack), the payload is stored by the system, and may later be embedded by the vulnerable system in an HTML page provided to a victim. The attack is carried out when a victim visits this page, or the part of the page on which the payload is stored.

Usually, to prevent untrusted code from gaining access to content on other domains, protection mechanisms such as sandboxing are applied or the same origin policy is enforced. However, XSS attacks bypass the same origin policy to gain access to objects stored on different domains by luring the victim to download or execute malicious code from a trusted site. Beyond sandboxing, the most commonly employed defense against XSS attacks is input validation [3, 19, 8, 11]. This approach uses a server-side filtering module that searches scripting command or meta-characters in untrusted input, and filters any such content. If the server fails to filter the input, however, the client is left defenseless. Other popular input validation techniques include dynamic tainting and untrusted information tracking. As highlighted by some recent work, these solutions correctly track whether a filter routine is called before untrusted information is output, but they do not reason about the correctness of employed filters, and fail to consider the Web application output [4]. Further, there are many scenarios where filtering is difficult to carry out correctly, especially when content-rich HTML is used. For example, attacks that are launched by scripts located at multiple locations in a Web application may succeed. A single filter function may not be sufficient if it looks for scripting commands, as injected input may be split across the output statements. In this case, every character in the HTML character set is legal, which implies that the filter cannot reject any individual character that may result in script content. Unauthorized scripts can be obfuscated by entering it within pre-existing execution environments, allowing it to escape the filters. That is, the attacker may embed an environment variable in between two existing tags. Hence, one should check the alteration of the execution flow to identify such hidden attacks.

3 Our approach: the XSS-Dec

Among the most popular techniques for Web vulnerabilities, anomaly detection and control flow analysis have gained popularity in the recent years. When considered alone, neither approach is however sufficient for effective detection of XSS attacks. First, the complexity of XSS attacks prohibits any approach that solely relies on anomaly detection [17]. Anomaly detection is in fact unable to detect most XSS subtle attacks, that are often deployed by exploiting obfuscation techniques. For example, XSS attacks are

written using JavaScript or ActionScript, where the output of the script is dynamic in nature, thus obfuscating the attack. Second, control flow analysis is effective for detecting subtle attacks, but it is very inefficient for real time detection [10] as it is slow and results in a high number of false positives. Real time detection is important with XSS attacks since the output of the attack script is often developed on the fly.

In order to overcome these limitations, we have devised a hybrid solution that combines the benefits of control flow analysis and anomaly detection to protect client systems against XSS attacks. Specifically, we suggest a security-by-Proxy solution, referred to as XSS-Dec. XSS-Dec relies on a Proxy component for vulnerability analysis and detection. The Proxy acts as a middle-man between the servers of the sites visited by the client, and a client-side Plug-in. This design upholds users' browsing activities while thoroughly monitoring the sites' vulnerabilities *before* any attack is carried out.

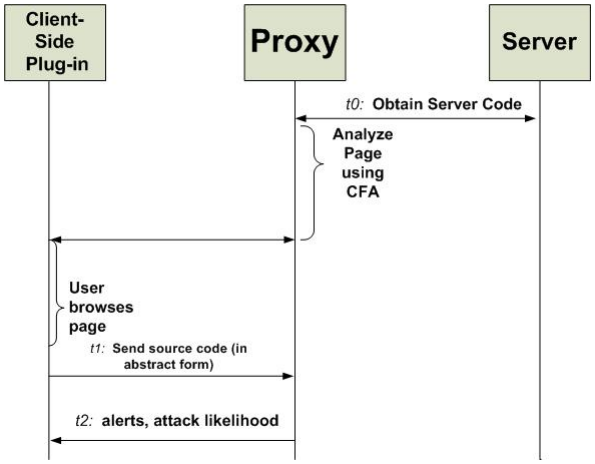

Fig. 1. XSS-Dec main flow

An overview of the XSS-Dec main functionality is reported in Figure 1. As shown, a first bootstrapping step is executed at time t_0. The server (or servers, if more than one is in fact connected to the Proxy) sends the encrypted copies of the source files of its Web pages to the Proxy. Subsequently, as the source of a Web site is updated or changed at the server end, more updates are sent to the Proxy by the server (possibly before the newer site is launched to the public). We assume that the Web sites' source codes collected by the Proxy at this point of time are valid, that is, there has been no chance for an attacker to insert any malicious script. The Proxy generates an abstract and accurate representation of the site, using control flow analysis, and stores it for later use. When the user starts browsing (at any point of time $t_1, t_1 > t_0$), the client-side Plug-in deploys the actual defense mechanism. Precisely, the Plug-in keeps a local record of the pages visited by the user. Further, it communicates to the Proxy the client input and the source code of the Web -page as it appears to the client. Upon receiving the client-end input, which is again encoded using control flow techniques, the Proxy

detects whether there exists any features indicative of malicious code or script, at time t_2, using both anomaly-based and signature-based detection.

Using signature-based detection, the Proxy searches and extracts features, which it uses to calculate the likelihood of an actual attack taking place. This attack likelihood is used to drive the Plug-in to either work pro-actively by blocking certain user actions and sites, or reactively by waiting for the attack to actually take place before notifying the user. This information regarding the attacks is sent back to the Plug-in at time t_2. The Plug-in, using the information obtained from the Proxy, deploys the actual defense mechanism, by either stopping the attack or preventing it from being executed (time t_3).

Note that the server and client side representations of the page being compared are different: the server source code is free of any injected malicious code, while the input received from the client-side may include malicious content.Although the client's actual input actions may differ from those simulate on the server side, injection of malicious scripts always result in a particular set of code features, like certain HTML tags being manifested in a compromised site. These features are the ones analyzed by the Proxy.

It is worth noting that our security-by-Proxy design assumes that the Proxy is resistant to basic attacks. The servers of the sites frequented by the client are assumed to be semi-trusted, and able to send to the Proxy non-corrupted data. That is, we trust the server to send the source code of its Web sites to the Proxy before the malicious scripts are injected. In line with current solutions based on client systems (e.g. [13]), we also assume that the Plug-in is not compromised.

4 The Proxy

The core algorithms behind our defense mechanism reside at the Proxy. The Proxy is composed of two logically distinct modules, Calculator and Analyser. These two modules serve the complementary tasks of analysis and detection.

4.1 The Calculator

The Calculator is in charge of computing a normalized and detailed view of the server's pages' content. The source code for both client and server's pages are modeled through a control flow graph (CFG), for accurate and efficient computation. The CFG is an abstract representation of the source code of the Web page, including any possible redirections for the URLs contained in it, and execution paths for active components, such as JavaScript or Flash components. In the context of our system, the CFG is represented as a directed graph. The nodes represent either HTML tags or actual program statements and variables. The edges represent the paths of execution, while the directions are dictated by the loops and the conditions present in the code.

CFGs have been often successfully used in static analysis [13, 5]. However, given the complexity of certain pages, CFGs can be computationally expensive to generate, and hard to navigate. This makes it very difficult to construct the dynamic CFGs for such pages on the fly, which is essential to identify the possible malicious effects of any code that has been added to the page. Our challenge is then to compute CFGs that are both accurate and efficient for the scope of our detection. To cope with these limitations,

we construct different types of CFGs, based on the specific Web page structure and of its content, as specified below.

1. *Page with no active components:* If a page has no active components, its CFG is derived from the control flow information available from the design model of the Web page such as its HTML or XML [18]. Specifically, the Abstract Syntax Tree (AST) [18] is first created, and then any flow information between the nodes is added. In what follows, this CFG is also referred to as model-based.
2. *Pages with active components:* If a page has a lot of active components, the CFG is again derived from the control flow information available from the design model of the Web page, and it is then augmented with the control flow information available from the actual code about active nodes. In particular, Flash-based elements and JavaScript components are expanded to uncover potentially obfuscated attack code. That is, when a $\langle$`script`$\rangle$ tag, or a $\langle *$`.swf`$\rangle$ file is encountered in a node of the model-based CFG, the node is further expanded based on the component's source code (i.e., the JavaScript, or the ActionScript respectively), and a new sub-CFG is obtained. The new CFG is constructed by representing each command in ActionScript in the code as a node. The flow from one statement to the next is given by directed edges. Notice that for the construction of this sub-CFG (i.e. the one containing expanded active nodes), we do not consider the user's inputs. Instead, we construct all possible execution paths based on all the possible inputs. Therefore, the CFG shows the call relations at the basic block level, while also containing all the possible nodes and edges. An example of a portion of an enriched CFG is given in Fig 2.
3. *JavaScript Rich Pages:* If active components are only JavaScripts, a simpler form of CFG is generated, to save both space and time complexity. Instead of generating the augmented dynamic control flow graph as described above, the JavaScript elements are rendered as augmented ASTs. The grouping parentheses (such as $\langle$`script`$\rangle$ tag, or a $\langle *$`.swf`$\rangle$) are still left implicit in the tree structure, and the syntactic representation of any conditional nodes are represented using branches, but the call relationships at the block level are still explicitly shown. Therefore in the augmented AST, the nodes are used to represent the commands in the code like in a simple AST, the loops are simply represented by if-then clauses with a given set of steps repeated in between. Any goto statements are also simplified to if-then-else clauses. Any user actions that can alter the loop (e.g. open a new page, click a link, move mouse over some objects of the page) are represented on the edges.
4. *Access Restricted Pages:* The CFG for a site which is access restricted (requires a login to gain access to portions of the site), is developed using a different methodology. Clearly, the site structure and corresponding CFG depend on whether the CFGs are built before or after the user's login. Further, the CFG for each user will be different as users may have customized Web spaces within the site. In case of such access-restricted sites, only the CFG before login remains the same across all users. The Calculator can easily obtain the CFG for this portion of the site. This CFG is very important, as any injected code on this page can potentially allow the attacker to take over the user accounts. Yet, damaging script can be injected in the pages after login too. To compute the CFG of user-restricted portions of the sites, the Calculator logs in using a test account. Intuitively, this CFG will not contain

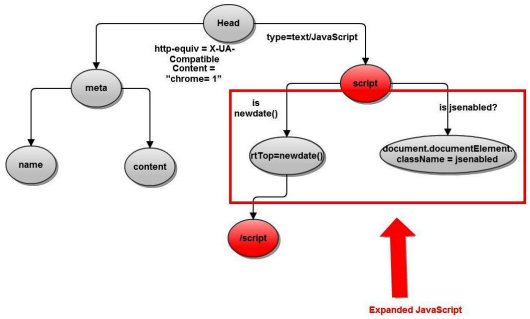

Fig. 2. Portion of the model-based CFG for the Yahoo site

user-specific information. It is however still useful for attack analysis, in that it gives the actual structure of the pages of the site, thus allowing the Proxy to detect any attacks that are launched by modifying the structure of the site. In particular, it helps detect any changes to the DOM structure and is therefore useful in detecting Persistent or DOM-based vulnerabilities. However, it cannot detect non-persistent vulnerabilities, which form the most common type of XSS attacks. This is because all the non-persistent vulnerabilities are exploited by data injected in the user-specific pages when the site is a login-based site. To detect the non-persistent vulnerabilities, we depend on the feature extraction capabilities of the Analyzer as explained next. If the Analyzer encounters a JavaScript or ActionScript environment, it requests the Calculator to compute the sub-CFG for that particular portion of the site, to detect possible malicious code injected within these environments.

4.2 The Analyzer

The Analyzer has two main tasks. First, it extracts features indicative of potential exploits. Second, it estimates the likelihood of the attacks being carried out.

Feature Extraction: The Analyzer, upon obtaining a client-side CFG, compares the client-side and the server-side CFGs to extract relevant features that may be indicative of attacks. In the following, we provide a broad classification of the features searched by the Analyzer. The features refer to non-access restricted sites, and are presented in the order of the severity of the attack.

(1) **Redirection to a site not contained in the server-side CFG**: If a CFG generated at the client contains a redirection to some site not contained in the server-side CFG, it likely means the user will be redirected to a site unknown to the original server. This feature, which is the most common and strongest indicator of an XSS attack, is often observed in DOM-based attacks [7].
(2) **SQL Injection Via XSS**: A script capable of inserting input on behalf of the user is potentially indicative of an attack. Specifically, if the script is added to the site without any actual action or permission from the user, and therefore appears to the

client-side CF , it may denote a *SQL Injection attack*. The SQL statements are used to commit changes to the database on the victim's behalf. Given below is an example of the code:

$<$ `TableID` $=$ "`TNAME`" $>$
$<$ / `Table` $>$
$<$ `Script Language` $=$ "`JavaScript`" `src` $=$ "`addjscript.js`" $>$
$<$ `Script Language` $=$ "`JavaScript`" $>$
`sql`("`insert` + `TNAME` + `values`('`Victim`', '`pwnd`', '`again`')");
$<$ /`Script`$>$

This script attempts to insert the values "Victim", "pwnd" and "again" into the table named TNAME. Using such statements, the attacker can change the passwords or other information of the victim.

To identify potentially malicious actions, the Proxy specifically monitors for server-side database actions being committed through SQL commands such as "UPDATE", "DELETE" etc. That is, it scans the JavaScript and ActionScript for any embedded SQL queries as in the above example. The Proxy also checks the CFG for all possible SQL commands including "SELECT", so as to identify a large range of attacks. This feature occurs often, though not exclusively, in stored attacks [20].

(3) **JavaScript based manipulation on the client-side CFG**: If the client side CFG includes nodes with $<$`submit`$>$ and $<$ `META` $>$ tags, forms may be submitted on the user's behalf, or cookies may be manipulated without the user's knowledge. Although each of these tags can occur for legitimate purposes in non-malicious JavaScript, when combined with any of the other features (especially redirection to a site contained on the server-side), these tags are typically representative of an attack. This feature is most often observed with reflected XSS attacks [7].

(4) **Text changed from original server site to the site rendered at the client**: Differences in the way text is rendered on the client's browser versus the way it was stored on the server-side are also to be treated as a warning sign. To check for any changes, the Proxy looks for alterations in the text formatting tags such as the $<$`header`$>$ tags, the $<$`para`$>$ tags, the use of bold or emphasis tags, etc. In this way, the Proxy can detect subtle attacks, where the attacker simply changes the way a Web site looks with intentions of slander or misrepresentation. Text manipulation can be carried out by any type of XSS attack, but is most commonly observed with DOM-based and reflected XSS attacks [16].

For access restricted sites, extracting the features of a latent attack is more complicated, as comparing the client-side and the server side-side CFGs is not sufficient. This is because the server-side CFG is derived using login information different from the login of the user being monitored. The CFG at the server side, while structurally similar, does not contain the actual information contained in the client-side CFG. For instance, the server-side CFG for a GMail page is constructed using a login different from the login used by the user, and therefore it has the same UML structure as the client-side CFG, but it will differ with respect to the exact content in the CFG.

For such sites, the Analyzer exploits the similar CFG structure of the two versions of the site to identify if the basic representation of the page is altered. In this way, text changes to non-user generated texts such as logos can be detected in the same way

as it was for non-access restricted sites. Further, the SQL Injection feature also does not change, as identifying SQL statements which cause actions to be committed on the user's behalf can be detected without the need for comparing the client-side and the server-side CFGs. Yet, the attacker can still inject the malicious script in those portions of the page that are actually user-specific. Referring again to our GMail example, the attacker would insert a URL for redirection in the actual mail content. To address this, the Analyzer checks whether any of the URLs that appear in the user-specific portions of the page link to a potentially malicious page. JavaScript manipulations are also hard to extract for access restricted pages, as the manipulation of the JavaScript can take place within pre-existing environments, which occur only on the user specific portion of the page. For example, the attacker can inject malicious scripts between two $<$ `META` $>$ tags present in the user's profile on a Facebook page for a link to his personal Web site. Since these $<$ `META` $>$ tags will vary for each user's Web site, there is no way of identifying whether any code has been injected between them by simply comparing the client-side and the server-side CFGs. Therefore, the Analyzer requests the Calculator to derive the CFG for the user specific parts of the page. Based on this input, the Analyzer checks for any possible malicious execution paths.

Attack Analysis. The Analyzer, upon detecting one or more of these features, computes the likelihood (denoted as α) of an actual attack. Each of the features is assigned a weight. The weight is set to correlate with the amount of past security-relevant information about the feature including the frequency of mentioning in incident reports. For example, the most common condition observed in XSS attacks is redirection or some sort of URL submission [7]. Therefore, the feature of redirection to a site not contained in the server-side CFG is to be assigned a high weight. α is simply calculated using the weighted sum: $\alpha = \sum_{f=1}^{|F|} w_f * n_f$. In the equation, w_f represents weight of the feature f, and n_f indicates the number of times the feature has occurred on the page. $|F|$ is the cardinality of all possible features the Proxy analyses. The equation can be extended to capture additional features, and strings which may be injected by the attacker. For example, a combination of features or a specific pattern of extracted features can be given an additional weight, or an additional attack string can be considered in the equation. For simplicity, however we stick to the formula specified above. As shown in Section 6, it is sufficient to guarantee a very good detection rate.

α is matched against anomaly based thresholds. We consider two threshold levels, a *detection* threshold and a *prevention* threshold. These thresholds are dynamic thresholds in that they are constructed based on the actual set of features which is extracted from a page. The *detection* threshold is indicative of a suspected attack, in that a number of limited features are verified true. It is therefore set based on the total number of features that have been extracted for a given page, and the lowest weight found in the set of features extracted. Using the lowest weight found ensures that the evidence presented to the Plug-in is just enough to register some suspicious but not necessarily harmful activity. The second threshold, which is is instead higher, models cases where there is enough evidence (in terms of features occurrence and importance) to believe that the attack is in fact imminent. This threshold is referred to as *prevention threshold* as it triggers mechanisms to prevent an attack, upon being passed. It is, also set using the total number of features extracted. However, it is based on the highest weight feature

that has been extracted for the given page thus making it higher than the detection threshold. For example, page X may contain added redirections to URLs, JavaScript manipulations and text changes from the original page on the server, while page Y may contain redirections, SQL Injections via XSS and JavaScript manipulations. The highest weight extracted for both page X and page Y is the same. However, the lowest weight extracted for page X is the weight associated with text changes, and the one extracted for page Y is the weight associated with JavaScript manipulations.

5 The Client-Side Plug-in

The client-side Plug-in is in charge of providing the Proxy with information about the pages the user is visiting, encoded as a CFG. Further, it deploys protection mechanisms against latent or ongoing attacks, upon being notified by the Proxy.

To complete these tasks, the Plug-in has two main modules: the *Auditor* and the *Detector*. The Auditor obtains information from the Proxy about all the possible attacks on an open page, while the Detector is in charge of stopping or preventing the identified vulnerabilities from being carried out. In order to help identify the possible attack vectors of an open page, the *Auditor* keeps a record of the pages visited by the user, and calculates a model-based CFG for each of such pages. Once created, each CFG is stored in an encrypted form. The CFG is subsequently updated or replaced as needed, according to any changes of the page's code, due to script injections or server-side modifications. The Plug-in sends the latest encrypted CFG to the Proxy, every time the user visits the page, as soon as it is opened on the browser.

The other module of the Plug-in is the *Detector*. The Detector is the component obtaining instructions from the Proxy if the features detected are deemed indicative of a potential attack. Precisely, it receives information about the possible attacks in the form of the attack likelihood α and the specific features extracted by the Analyzer. Each of the features results in a particular type of anomaly. The anomalies consist of execution of a script on a site to which the user has been redirected, personal information of the user being sent to a remote system, and actions such as submission of forms taking place without any corresponding input on the user's part. The first two anomalies are often observed in case of redirection and JavaScript based manipulation, while the third one is often observed with SQL Injection via XSS.

The Detector monitors the client machine for any of such anomaly using dedicated modules. Each module corresponds to a specific monitoring activity and is activated if the Proxy verifies the feature they implement. Specifically, the module checks for specific user actions which actually start the attack based on the features it has detected. If one of the extracted attack features consists of unwanted redirections (see feature (1) in Section 4.2) and the estimated likelihood is high, the Detector prevents the user from being redirected to the targeted page. Otherwise, (i.e. likelihood is below the prevention threshold) the Plug-in simply pops up an alert box to the user when the link is opened. In case an attack presents one or more features (i.e. feature (2), (3) and (4)) beyond redirection, and has an estimated high likelihood value, the Detector prevents the malicious script from being executed. Specifically, it does not render a portion of the page or an entire page and displays an error message to the user. In case of a low likelihood value,

it simply pops up an alert to the user before rendering the malicious content. Intuitively, stopping ongoing attacks is less desirable as preventing them. As shown in the next section, the XSS-Dec is efficient enough to stop the attack before any major damage of the client system.

6 Evaluation

We deployed and thoroughly tested a running prototype of the XSS-Dec. Before discussing our evaluation, we briefly describe the prototype developed.

6.1 XSS-Dec Prototype

The Plug-in was developed by logically distinguishing the Auditor from the Detector. Both components were implemented primarily in JavaScript, to guarantee portability. The Auditor uses a separate JavaScript component to construct the CFG based on the HTML and DOM structure of the page. Each of the possible user actions are edges' labels. In case the node contains a URL (i.e., a `href` tag) to some other site or page, this node becomes the second node of the CFG of that page. To reduce the risk of interceptions of the user's browsing history, the Plug-in sends the CFG in an encrypted form. In the current prototype, the Plug-in uses Merkle hash trees [14] for easy graph comparison and reduced graph size. Alternatively, we could serialize the tree and rely on more traditional encryption schemes.

The Detector is organized into four JavaScript modules, one for each of the features possibly extracted by the Proxy. Each module is activated if the Proxy verified true the corresponding feature they implement.

The prototype of the Proxy also has two modules. Both are implemented using Java and JavaScript components. Specifically, the Calculator uses JavaScript to build the nodes for the model-based CFG. Java components are then used to expand the active nodes, i.e. the nodes containing JavaScript or Flash elements. The resulting enriched CFG is built as a serialized tree using the TreeMap Java class. The Calculator stores a number of CFGs of the pages being most often visited by end users. This simplifies both CFG analysis and comparison with Plug-in-received CFGs. The stored CFGs are updated as the pages' content is notified to have changed.

6.2 Experimental Evaluation

The goal of our evaluation was two-fold. First, we aimed at estimating the accuracy of our solution in detecting XSS attacks. Second, we estimated the overhead incurred with our protection mechanism. Estimating the Proxy overhead allows us to make some initial considerations on the scalability of our solution. The Plug-in was tested from a Dell Latitude D630 Laptop, with 2G Ram and a Intel(R) Core(TM)2 Duo CPU T7500@ 2.20GHz processor. The Proxy was run from an Apache server hosted on the same machine, to maintain a conservative estimate of the system efficiency. The server was running the Apache Web server (version 2.2) and PHP version 5.3.3. Apache was configured to serve requests using threads through its `worker` module. Our tests do not

account for any network delays, and are carried out without conducting any fine-tuning or training.

Detection Accuracy

Experimental Settings. Using a trial-and-error approach, we defined two simple anomaly threshold values for assessing whether an attack was latent or not. We express the features' weights of equation in Section 4.2 by means of totally ordered integers, ranging from 1 to k, where $k \leq |TF|$, and $|TF|$ is the possible total number of features (4, in our case). Given a set of extracted features $F = \{f_1, \ldots, f_n\}$, we compute the prevention threshold as follows: T_{high}=$|F| + w_{f_{max}}$, where $w_{f_{max}}$ is the highest weight among all the weights of the extracted features. $|F|$ represents the total number of features extracted (regardless of their actual weight). If a same feature appears more than once, it is counted as a new feature, therefore increasing the overall probability of an attack. Intuitively, from this equation, we can determine whether the feature of highest weight is significant enough to influence α to a point where an attack is most likely to happen. Our detection threshold, referred to as T_{low} is computed in a similar fashion of T_{high}: $T_{low} = |F| + w_{f_{min}}$ and $w_{f_{min}}$ is the lowest weight assigned to the features in $\{f_1, \ldots, f_n\}, n > 1$. When $T_{low} < \alpha < T_{high}$, the Proxy suspects an attack. It sends a warning message to the client Plug-in, providing details about the warning features verified true. When $\alpha > T_{high}$, the Proxy deems that the likelihood of an attack is very high.

Methodology We evaluated our system on several real-world, publicly available Web applications and on simulated environments. We recorded the number of false positives generated when testing the application with attack-free data and the number of attacks correctly detected when testing the application with malicious traffic. In detecting the attacks we tracked whether they were detected at the time of prevention (i.e. α was above T_{high}) or detection (i.e. α was above T_{low}).

Overall, we ran the XSS-Dec system for a total of 100 pages, in a non-deterministic order. 20 of them were hand-evaluated real-world clean pages. The remaining pages were constructed by us, and contained one or more XSS vulnerabilities. The clean pages were selected from popular Web sites with active components, like MSN, Yahoo, Google, social networking and forum sites. The vulnerable pages were created using the real-world XSS vulnerabilities reported in the security mailing list, Bugtraq [2]. We deployed the given vulnerabilities in similar sites than those listed as vulnerable, and injected the malicious script in the variables described. We constructed 80 sites, and tested 80 different vulnerabilities. Each of these sites hosting the malicious files had benign components. The actual attack code varied for each try, so as to create polymorphic attack code. To create the variations of the attack code, we introduced random NOP blocks in each attack to introduce random delays. Further, we combined one or more attacks with each other, i.e. some vulnerabilities were tested multiple times. Also, the page invoking the malicious content was different for each try. The elements we included in each page consisted of one or more of the following: images, videos, audio components, other benign JARs carried in applets but not embedded in images, text documents, hyperlinks, Java components, JavaScript components, forms, zip files, Microsoft Office Open XML documents, XPI files,benign SWFs and simple games.

Attack Type	Detected	Prevented	False Positive	False Negative	Total Attacks
Normal XSS	0	All	0	0	15
Image XSS	0	All	0	0	13
HTML entities	0	All	0	0	12
Style-Sheet based XSS	0	All	0	0	13
Flash-Based XSS	5	3	0	2	13
XSS in pre-existing environments	0	All	3	0	14

Table 1. Evaluation Results

Results Table 1 summarizes our results organized according to the classification in the Rsnake Cheat Sheet [7]. The results reported in the table group the different 80 attacks according to the location of the attack vector. As shown, XSS-Dec stopped all but 2 attacks. Both were Flash-Based. Out of the stopped attacks, 94% of them were prevented before being carried out. The remaining 6.2% were stopped at detection time. We reported 3 false positives. A false positive occurred when an attack was detected in a part of the page where there was no attack code.We noticed that false positives were detected on the forums of users sharing coding tips on JavaScript. The code displayed on the pages as part of the discussions was considered as an injection by the XSS-Dec. To improve the false positives on forums, we plan on adding string checking to the Proxy as part of our future work. String checking will help differentiate between the code being discussed in the forums and some malicious script. The sites that were not prevented but only detected were typically sites with a huge number of Flash components. Flash components enable the attacker to hide the consequences of the redirections due to script injection, reducing the overall likelihood of the attack being prevented by the Proxy. We expect that training the model would mitigate these issues. Below, we summarize our results for three of the most challenging categories of attacks: In case of Flash-Based attacks, our approach prevents most of these attacks by not executing the Flash file. For those attacks that are only detected, the file is executed but the user is alerted as soon as some malicious activity is seen on the client end. In case of Cookie stealing XSS attacks, our approach specifically monitors for manipulation of the ⟨`META`⟩ tags to reset the cookies, and detects all possible instances of this vulnerability. For XSS attacks where the vectors are injected into pre-existing elements (e.g. between pre-existing ⟨`script`⟩ and ⟨/`script`⟩ tags), our approach monitors for manipulation of JavaScript and we achieve a 100% prevention rate.

Performance Evaluation

We computed the average time for the most resource consuming activities of our system, i.e. constructing the CFGs and extracting features. Our tests show that the average time for constructing a CFG of level 40 with no dynamic components is 3.25 seconds, and for constructing the CFG with 50% dynamic components is 3.39 seconds. The time grows linearly with respect to the size of the CFG. For these tests, we used CFGs of increasing complexity from 10 to 80 nodes, each corresponding to real sites. The CFGs of level 80 correspond to popular sites, with a large number of active components (3/4 of the nodes), such as Youtube and Bigfish. The complexity of the CFGs increased as the ratio of active nodes to inactive nodes increased. For the simpler CFGs, the ratio of active nodes to inactive node was 1:4, while for the more complex CFGs,

the ratio was greater than 3:4. The highest complexity for a CFG of level 80 was 83% of the nodes were active nodes. The time taken for constructing the CFG of level 80, with 83% active nodes was 4.183 seconds, while the time taken to calculate the least complex CFG was 3.2432 seconds. This makes the overhead for the most complex CFG compared to the least complex CFG less than 1 second different. We notice that while this time is not negiglible, CFGs are only calculated periodically, and cached for efficient reuse.

The time for extracting features on an average for a CFG of level 40 is 2 seconds, while the maximum time for a level 80 CFG is 2.673 seconds. Since the CFGs are computed at the Proxy, these results confirm that the Proxy is indeed scalable. In real-world settings, the Proxy would be hosted on a dedicated server with larger processing power than our system. Further, we notice that since most of the pages maintain a similar structure the Proxy can improve the size of the cached directory of model-based CFGs, for similar Web sites. This would likely improve the performance substantially. Finally, the Proxy in the real world would not be running in parallel with the Plug-in as was the case for our system.

7 Related Work

XSS attacks have been identified as a threat since the 1990s. Since then, various solutions to detect and prevent these attacks have been explored. Traditional solutions focus on sanitizing the input at the server side, but recently client side approaches have also been proposed. There also exist Proxy-based solutions which aim to protect Web applications by analyzing the HTTP requests. Despite these efforts, XSS attacks still remain on the top of Web security attacks in the OWASP lists [15].

Server-side solutions or Proxy-based solutions are commonly used for Web -based attacks, since they enable users' inputs sanitization [3, 19, 11, 22]. In particular, Scott and colleagues proposed an interesting Proxy-based solution [19]. The Proxy is similar to an application firewall; it enforces pre-written security policies. Their proposed mechanism requires that all Web applications patch themselves to prevent an XSS attack. In case a Web application is not patched, the end user is left defenseless. Our focus, on the other hand, is how to ensure that any malicious script does not affect the user. Consequently, it does not require any patching from either the user or the server. Similar to the above is a commercial product called AppShield [1]. AppShield also inspects the HTTP messages to prevent application level attacks. While it is similar to our system in inspecting the HTML of the pages outbound from the server, it does not specifically look for any code injection. Hence, Appshield can recognize attacks based on the (proprietary) rules that it uses to validate the HTTP requests. Wurzinger also propose a proxy-based solution to detect HTML responses and any injected scripts [23]. To identify malicious scripts, any legitimate script calls in the original web page are changed into unparsable identifiers called script IDs. Therefore, if any unparsed script is found, it is assumed ti be indicatory of an attack. This system focuses on stored and reflected XSS but not on DOM Based attacks. Further, the parsing of a script may be a significant bottleneck of the system.

Bisht et al. [4], propose to remove any server side script in the output of a Web application, when the script is not originally inserted by the application itself. This approach

is complementary to ours in that we focus on preventing the attacks at the client-end, rather than relying on servers' filters only. Further, as any other server-based solution, Bisht's approach relies on the server ability to patch and remove server side scripts. The fact that a solution focusing on protecting the servers may leave end-users vulnerable has inspired some interesting client-oriented solutions. One of these is the Noxes system, proposed by Kirda et al [13]. Noxes is a Web firewall aiming at protecting the client from XSS attacks. Noxes' detection is based on the analysis of the server-side scripts. In XSS-Dec, we also use server-side scripts. However, our detection of code injection relies on a detailed comparison of the server-side scripts with client-side scripts. Kirda and colleagues instead choose to rely on validating the HTTP referrer headers. The HTTP headers, however, do not represent a useful indicator in case the attacks come from trusted sites. Further, the information leaked via embedded URLs is contained by limiting the information sent through each.

We borrowed the idea of using control flow analysis from some recent interesting work [9, 21, 5, 6]. The Swaddler system [9], for example, focuses on detecting any violations in the workflow of a stateful application or input violations by users. We differ from this work both in scope and in the detection mechanism: our focus is on script injections rather than state violations. Further, our solution accounts for both stateful and stateless applications. Bonfante et al. in [5] used control flow graphs for extracting malware signatures. The authors present a system for extracting signatures of malware by using CFGs composed at the assembly language instruction set level. While similar to our approach in spirit, our CFGs are derived based on high level languages. We employed control-flow analysis in our previous work, the DeCore [21]. The DeCore is aimed at detecting content repurposing attacks, from the client-side end, and therefore focuses on a different set of attacks. Close to the notion of control flow analysis is script analysis, which has been leveraged to detect XSS vulnerabilities. A specific example of this approach is the Pixy tool proposed by Jovanovic et al.[12]. We take a complementary approach, in that we analyze JavaScript, ActionScript and HTML. Further, the Pixy tool relies on taint analysis of the data whereas we leverage the notion of control flow analysis by using CFGs. The CFGs allows the XSS-Dec to detect any malicious script injection using any type of script, while the taint analysis in the Pixy tool helps detect any input violation.

8 Conclusion

In this paper, we presented XSS-Dec, a security-by-Proxy approach to protect end-users against XSS attacks. Our solution combines the benefits of both server-side and client-side protection mechanisms. We leverage the information obtained from both the client and the server-side to provide an anomaly based detection approach complemented by control flow analysis. In the future, we will study whether a server can use the Proxy features without having the server's sending pages beforehand. Finally, we will test the scalability of the XSS-Dec in distributed settings.

References

1. Appshield, 2004. Sanctum Inc.

2. Security focus -bugtraq, 2010. http://www.securityfocus.com/archive/1.
3. D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. In *19th international conference on World Wide Web*, WWW '10, pages 91–100. ACM, 2010.
4. P. Bisht and V. Venkatakrishnan. XSS-guard: Precise dynamic prevention of cross-site scripting attacks. In *5th GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assesment*. Springer, 2008.
5. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Control Flow Graphs as Malware Signatures. In *International Workshop on the Theory of Computer Viruses TCV'07*, Nancy France, 2007.
6. S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y. min Wang. A systematic approach to uncover security flaws in gui logic. In *IEEE Symposium on Security and Privacy*, pages 71–85. IEEE Computer Society, 2007.
7. ComputerWeekly.com. Hackers broaden reach of cross-site scripting attacks, 2007.
8. S. Cook. A Web developer's guide to cross-site scripting. t. r, SANS institute, 2003.
9. M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Recent Advances in Intrusion Detection*, volume 4637 of *LNCS*, pages 63–86. Springer, 2007.
10. C. Earl, M. Might, and D. V. Horn. Pushdown control-flow analysis of higher-order programs. *The 2010 Workshop on Scheme and Functional Programming*, 2010.
11. M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Annual Network & Distributed System Security Symposium*, 2009.
12. N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE Symposium on Security and Privacy*, pages 258–263. IEEE Computer Society, 2006.
13. E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *2006 ACM symposium on Applied computing*, SAC '06, pages 330–337. ACM, 2006.
14. J. L. Munoz, J. Forne, O. Esparza, and M. Soriano. Certificate revocation system implementation based on the merkle hash tree. *International Journal of Information Security*, 2:110–124, 2004. 10.1007/s10207-003-0026-4.
15. OWASP. Top 10 2010 - the open web application security project, 2007. www.owasp.org.
16. OWASP. DOM based XSS, 2011. https://www.owasp.org/index.php/DOM_Based_XSS.
17. P. Raman. JaSpin: JavaScript Based Anomaly Detection of Cross-Site Scripting Attacks. Master's thesis, Carleton University, Ottawa, Ontario, 2008.
18. N. Schwartz. Steering clear of triples: Deriving the control flow graph directly from the Abstract Syntax Tree in C programs. Technical report, New York, NY, USA, 1998.
19. D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of the 11th international conference on World Wide Web*, pages 396–407. ACM, 2002.
20. SpiderLabs. Analysis of lizamoon: Stored XSS via SQL injection, 2011. http://blog.spiderlabs.com/2011/04/analysis-of-lizamoon-stored-xss-via-sql-injection.html.
21. S. Sundareswaran and A. C. Squicciarini. Decore: Detecting content repurposing attacks on clients' systems. In *Security and Privacy in Communication Networks (SecureComm)*, volume 50, pages 199–216. Springer, 2010.
22. G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *30th International conference on Software Engineering*, pages 171–180. ACM, 2008.
23. P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel. Swap: Mitigating xss attacks using a reverse proxy. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, IWSESS '09, pages 33–39, Washington, DC, USA, 2009. IEEE Computer Society.