STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 4, 2012

CROSS-SITE SCRIPTING BROWSERS' PROTECTION ANALYSIS

ADRIANA NEAGOŞ AND SIMONA MOTOGNA

Abstract. The purpose of this paper is to present an overview upon the protection methods browsers provide against cross-site scripting, as an important security vulnerability. Time has imposed different measures and competitors on the browsers market had to keep track. The study has three goals: to take into consideration versions of Internet Explorer, Google Chrome, Mozilla Firefox and Opera, to use an established scoring system, the Common Vulnerability Scoring System, to measure certain vulnerabilities on each browser, and to develop a tool for web application flaws testing.

1. INTRODUCTION

The success or failure of an application is characterized by three factors: quality, time and cost. Software development focuses on delivering applications with minimal resources (people, software components and hardware) and quality is not always set as an important issue. If time and cost are related to the project management and can be changed on the spot, quality assurance is a complex process that should not be neglected. Several studies (NASA [21], IBM [13]) have led to an important conclusion: improving quality reduces development costs.

Last years brought us in front of an explosion of web applications. Desktop products are replaced by the three layer architecture of the server machine, the client machine and the network as delivery mechanism. In these conditions, software quality factors are changing their importance, and security becomes an important factor because of the transactions that are made and because of the sensitive data that is sent over the internet. In order to ensure security, people need an evaluation scale of a software application and in consequence,

Received by the editors: October 30, 2012.

2010 *Mathematics Subject Classification.* 68N30, 68M11.

1998 *CR Categories and Descriptors.* C.2.0 [**Computer-Communication Networks**]: General – *Security and protection*; C.4 [**Performance of Systems**]: *Measurement techniques*.

Key words and phrases. Cross Site Scripting, Security.

a lot of research, both in academia and industry, focuses on studying risks, vulnerabilities, and attacks against security. Open Web Application Security Project (OWASP) [22] is such an initiative, and maintains and updates a list of top 10 security risks.

Cross-Site Scripting (XSS) is the second as importance, and considered as having an average exploitability and a high degree of occurrence. There are a lot of cases in which automatic tools detect this risk in an easy way, but, however, there are some situations, generated by new technologies and browser characteristics, that make detection more difficult.

The purpose of this article is to present an in-depth analysis of detecting and preventing cross-site scripting vulnerabilities and their impact on software security factor. The rest of the paper is organized as follows: Section 2 presents some of the existing related work. Section 3 contains an analysis of XSS vulnerabilities and the main prevention methods. Section 4 focuses on the security policies proposed by different browsers and contains a case study, while the next section is dedicated to our evaluation of some XSS vulnerabilities. The proposed tool is presented in Section 6, and in the end we draw some conclusions and future directions of study.

2. RELATED WORK

A lot of research has been carried out in the field of XSS vulnerabilities. Most of them focus on studying pattern attacks, evaluating risks and proposing solutions to prevent them [14], [26], [9], but as far as we know there is no paper comparing XSS vulnerabilities from the browsers' point of view. Release notes gather a sum of new features, but do not always point out the security issues and the user does not have a proper overview of what the measures taken imply and how does the situation look like considering the rivals on the market.

Regarding Security Evaluation and Measurement, discussed in the last part of our paper there are several approaches. Besides the approaches carried out at major software companies, such Microsoft, IBM, Apple a.s.o., there are two important contributions to assessing security vulnerabilities and proposing metrics to evaluate their impact on software quality:

- Computer Emergency Center (CERT) at Carnegie Melon University with results in risk analysis, based on a tactical and on a systematic approach, and security measurement, that are integrated in IMAF (Integrated Measurement and Analysis Framework) [6].
- Common Vulnerability Scoring System (CVSS) [17] that developed a framework that supports scoring of security vulnerabilities.

3. ANALYSIS OF CROSS SITE SCRIPTING VULNERABILITIES

Cross-site scripting vulnerabilities were discovered back in 1996 when web pages became more interactive due to a new programming language, Javascript and were associated with the name of Netscape Communications, the most popular browser at that moment. Even if it were first reported as Web browser vulnerabilities, David Ross demonstrated in his paper "Script Injection" [10] that the problems may also come from the server side rather than from the client. The vulnerability was first named "HTML injection" and then referred with the acronym "CSS", but the confusion with Cascading Style Sheets determined in 2000 a new convention, "XSS".

People tended to underestimate the power of XSS, because it could not damage the operating system or exploit a database, but the attack on October 2005, when the first major XSS worm, called the Samy Worm spread on over a million of MySpace accounts rose the public attention.

Javascript, ActiveX, Silverlight, or Flash are lightweight programming languages used for a friendly user experience and for a more dynamic interaction with the application. They provide code that is executed by the browser on the client and is the way XSS is performed. Browsers execute this kind of code under sandboxing mechanism which means that only a set of operations should be performed, but even this protection is not enough. Michael Howard said that "All input is evil until proven otherwise. That's rule number one" [11], but in case of XSS not only inputs, but also outputs must be validated.

There are 3 types of XSS:

- *non-persistent or reflected*: is performed when there is no proper validation of user input through GET or POST requests and the response page is returned immediately and is spread generally by email via malicious urls;
- *persistent*: happens when the infected code is stored in the database and it is a regular threat to chat software or application including different posts. User does not access malicious links, just regular browsing can result into being robbed of information;
- *DOM-based*: results from dynamically-computed data, which means that the browser is manipulated to render DOM elements controlled by an attacker.

The following is an example of reflected XSS in ASP.NET web forms. We consider a form used in a registration page in which the user is required to input some data and then submit. The page validates the input on the server side and then returns a message. Suppose the input requires an email address, but this address should not have been submitted before, so this is why the validation is done on the server and not on the client. If the email address is not valid an error message will be returned. The form looks like this:

$\langle form\, id = "form1" action = "\#" runat = "server" method = "get" \rangle$
  $\langle div \rangle$
    $\langle div\, id = "errorDiv" runat = "server" \rangle \langle /div \rangle$
    $\langle input\, id = "myinput" runat = "server" / \rangle$
    $\langle button\, onserverclick = "serverClick" runat = "server" \rangle Submit \langle /button \rangle$
  $\langle /div \rangle$
$\langle /form \rangle$

Now when the user clicks submit button, function *serverClick* is executed:

$\langle script\, language = "C\#" runat = "server" \rangle$
  $void\, serverClick(Object\, sender, EventArgs\, e)$
    $\{$
      $errorDiv.InnerHtml = "The\, following\, is\, not\, a\, valid\, email" +$
        $myinput.Value + "Please\, go\, to \langle ahref =' \#' \rangle Help \langle /a \rangle";$
    $\}$
$\langle /script \rangle$

If the user inputs some valid data everything works just fine, but XSS can be performed by sending the following input:

$\langle div\, onmouseover =' alert()' \rangle some\, valid\, input \langle /div \rangle$

Now that an attacker sees that the page is vulnerable he may change the javascript function executed on mouse over with some malicious code and send the URL asking for opinion on the received error. The web page seems trustful and the action is done on mouse over, so most of the users would not be suspicious receiving this.

We will use the same example for the stored XSS. Suppose the attacker has entered a valid email address, but he injected the following script in other input:

$\langle script \rangle window.location = "http : //maliciouspage.com" \langle /script \rangle$

In a normal scenario he should be redirected to a welcome page in which all the other users would be shown. Still, because of the script he inserted, script that was inserted in the database as he was careful to double the quote, the redirection page wouldn't be the welcome page, but his malicious site. Now, all the users after registration would be redirected to that page.

The example presented above is just a proof of concept. It is a personal ASP.NET web page, using Microsoft SQL Server 2008 R2 and tested on Internet Explorer 9.

Monitoring and improvement in preventing web applications against cross-site scripting follow the general steps used in avoiding vulnerabilities on web: code review, manual testing, automated testing and implementation of security policies on browsers, firewalls, each of them having pros and cons and assuring security to a certain extend.

3.1. **Code review.** There are two methods of reviewing code: a static analysis of the code when lines are reviewed without being executed or a dynamic analysis at the runtime. It is feasible for small and middle applications, but in case of complex products, it can be very laborious and time-consuming. It is difficult to verify all the tainted data if, for instance, code contains dynamic string-building techniques and predict the application's client state. Best practices advise white lists for the inputs and proper encoding for the outputs. Still, white lists are not always possible.

Any of these: Request.Params["input"], Request["input"], Request.Query String ["input"], Request.Cookies["input"], Session["input"], Application ["input"] can be sources of exploit and injection can be performed on tags like: ⟨ body⟩, ⟨frame⟩, ⟨img⟩, ⟨html⟩, ⟨layer⟩, ⟨link⟩ etc. or their attributes src, href, style.

Programming languages provide in-built functions that perform this kind of filters, but even Microsoft states regarding their ASP.NET method *ValidateRequest* that one should not rely only on this type of validation because unfortunately it is not 100 percent secure. Recent attacks prove this. Not only ASP.NET functions have security lacks. *parse_ url* is a function in PHP that verifies malformed urls. The function works correctly in most cases except if whitespaces are inserted. This was the vulnerability exploited on April 2011, on Facebook, when a malicious video was posted [25] or on CNN when urls inserted in ad networks were source of this attack. Other exploits were done also on The New York Times, on Twiter, e-Bay or on Fox News [15].

Microsoft offers an Anti-Cross Site Scripting Library [20] and OWASP advises programmers to use an API: ESAPI (The OWASP Enterprise Security API) [22] which is an open source web application security control library.

3.2. **Testing.** Testing any kind of input based vulnerability involves the same pattern: detection of a flaw, injection of data and confirmation of the exploit. A proxy can be used in order to intercept the HTTP traffic and TamperIE [4] for modifying the GET and POST requests.

Even if OWASP [22] classified it as easy detectable, we tend to disagree because of the variety of malicious strings: keyword ⟨script⟩ is not mandatory when code is inserted in body as attribute or event, quotes and double quotes can be alternated, whitespaces inserted or the statements can be encoded such that they are skipped by the filters such as:

$\rangle''' \rangle \langle script \rangle alert('XSS') \langle / script \rangle$

$\langle object\ type = text/htmldata =''\ javascript : alert(([code]);'' \rangle \langle / object \rangle$

$\langle body\ onload =''\ javascript : alert(([code])'' \rangle \langle / body \rangle$

There are several XSS Cheat Sheets to be used while testing, but the possibilities of injection seem infinite. Scanners and fuzzers were implemented to automate testing, Burp Suite [24], HP WebInspect [12], Acunetix [1] or Netsparker [16] are such applications. Mainly, they use the same mechanism: they insert a harmless string named XSS locator in the identified application data entries, scan the output and then perform the exploit in order to confirm the injection. The problem is that they test only using common scripts and technologies such as AJAX or JSON are not supported as they make this automation very difficult.

3.3. **Firewalls.** Runtime protection methods should also be taken into consideration, even if they affect the performance of the application. Web application firewalls(WAF's) monitor the communication across the network and therefore they inspect messages for Javascript and can enforce a set of rules in order to identify and to block XSS attacks that would not reach anymore the backend. Examples of such applications are Cisco ACEWeb Application Firewall [7], NetScaler App Firewall [8] or Barracuda Web Application Firewall [3]. Most WAF's implement the Intercepting Filter pattern or include one or more implementations in their overall architecture. One can also add filters to an application at deployment when implemented as a Web server plug-in or when activated dynamically within an application configuration.

Users should be also educated to avoid XSS exploits. Avoiding awkward links, paying attention to redirections or for instance turning off the HTTP TRACE can prevent the stealing of cookies.

Regardless the variety of prevention methods, new exploits continue to attack web applications and it's our duty to keep on protection against the known or unknown security flaws.

4. SECURITY POLICIES

We have intentionally omitted security policies from the above protection methods in order to provide a more in-depth analysis of this concept. XSS is the one security field that does not depend on the type of connection: encrypted or unencrypted, but is closely related to portability and mainly browser compatibility. Because it is rendered by different browsers, the display of a web page can be slightly different, and so its gate of access.

Same origin policy is called the policy adopted against browser-side languages that does not allow "access to most methods and properties across pages on different sites". This means that the sensitive information held about a certain user belongs and can be used only by the original site and cannot be accessed by other bad targeted sites. It is implemented by nearly each browser, but it does not guarantee complete security. In addition, modern browsers implemented several security policies that block an attacker to gain access on a client machine.

Firefox and Opera are known as relatively secure, while Internet Explorer (IE) is considered very vulnerable. A simple example is for instance when uploading text files through IE: if HTML content is inserted in the file, it doesn't treat it as plain text, but it interprets it as HTML. As this is not the most relevant example, we will try to make a fair analysis of the most used browsers, their evolution and the measures taken against XSS attacks.

Internet Explorer 6, Firefox 4 and Opera 9.5 introduced HttpOnly cookie attribute. It was intended to protect against retrieving information through document.cookie, but even with this, cookie information could be accessed through XMLHttpObject. Internet Explorer 7 made sure information was hidden and unavailable in the response header only if it was submitted from the same domain, but not the other browsers. "HttpOnly cookies don't make you immune from XSS cookie theft, but they raise the bar considerably" [2].

Fraud Protection was called the mechanism adopted by Opera 9.5 to enforce security. It was a tool powered with phishing information from Netcraft and PhishTank, and Malware protection from Haute Secure that provided automatic detection and warning of malicious sites and supported Extended Validation certificates.

Starting with Internet Explorer 8, a new very controversial browser filter for XSS has been introduced. It is similar to a proxy and the filtering is done using regular expressions. Still, Michael Brooks describes it as vulnerable and users claim that it also considers safe pages as potentially dangerous [5] and Google disables it by setting the X-XSS-Protection in the header to 0 or it can be turned off from the browser security tab. Similarly, Chrome 4 responded to the security features included in IE 8 and introduced a static analyzer which attempts to detect XSS, called Anti-XSS Filter. Adam Barth, a software engineer on the Chrome team, said "The XSS filter checks whether a script that's about to run on a Web page is also present in the request ... that's a strong indication that the Web server might have been tricked into reflecting the script"[1]. But even if it could have been considered a revolution in the security against XSS, methods to bypass the filter's protection and provide full disclosure appeared shortly.

Firefox 4 came with NoScript XSS Filter, an add-on that verifies the form of URLs. It reports if they have a suspicious content, but this without confirming the attack. It also added HTTP Strict Transport Security which informed the browser to automatically convert all attempts to access a certain site using HTTP to HTTPS requests instead. This was also supported by Chrome 4 and Opera 12 Beta, but not by Internet Explorer.

[1]from http://jetlib.com/news/tag/adam-barth/

In March 2011, together with the release of version 4, Firefox proposed the adoption of a new layer to enforce XSS protection called the Content Security Policy (CSP). This provides a way that helps the browser to differentiate between legitimate and malicious content. CSP requires that all JavaScript for a page are 1) loaded from an external file, and 2) served from an explicitly approved host. It means inline scripts are ignored unless they are defined in a white list. This framework is still not implemented yet on other browsers, but Microsoft claims that it will be a feature of Internet Explorer 10.

The methods stated above and adopted by the browsers are mainly applied against reflected XSS, as for the other two types of attack, it is not the duty of the browser to enforce protection.

4.1. **Case Study.** In the following lines, we will present a simple example of XSS attack against one of our web pages. It is an edit page that receives as parameter the ID of a document and then using this filtering retrieves information about that document and provides a way to modify the stored information (available at the url http://scs.ubbcluj.ro/ naie1000/phplab/editDoc.php). EditDoc.php?ID=1 is for example a valid request, but we have also tried inserting scripts. The simplest example

$\langle script\rangle alert(1)\langle/script\rangle$

was avoided in all browsers because of the position where ID parameter was inserted and because of the protection provided.

First script that was successfully executed was:

$'''-\rangle\langle/style\rangle\langle/script\rangle\langle script\rangle alert(0x000040)\langle/script\rangle$

So, let's see how different players on the browsers market react.

Internet Explorer provides a popup at the bottom of the page saying: "Internet Explorer has modified this page to help prevent cross-site scripting." and encodes characters in the url as follows:

$'\%22--\%3E\%3C/style\%3E\%3C/script\%3E3Cscript3E$
$alert(0x000040)\%3C/script\%3E$

Opera also makes this kind of encoding, but the url is not totally visible. The parameter is hidden and can be seen only if the user selects it on purpose. This way, an inexperienced user can easily be fooled and trapped in some attack of this kind. The alert appears on the page.

Firefox instead does no encoding and shows the alert without no validation.

Chrome also skips encoding, but the alert is not visible. The user is not informed about the blocked script as in the case of Internet Explorer, but it is protected against it.

To confirm the behavior we have tested using more different scripts and the four browsers reacted in the same way as above.

We used a harmless simple script just for example, but in the same way the alert is executed: any script requesting for instance document.cookie or redirecting the page to a malicious site can be inserted. We also used the default settings of each browser for a proper analysis of the vulnerability.

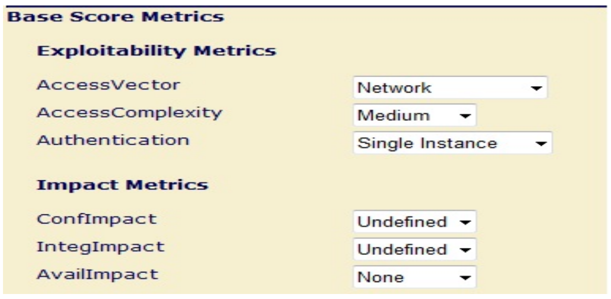
5. CVSS SCORES FOR XSS VULNERABILITIES

Our case study consists in computing the CVSS vulnerability scores for some XSS related vulnerabilities reported on the above mentioned browsers. CVSS or the Common Vulnerability Scoring System is an open framework that provides a numerical score by taking into consideration base, temporal or environmental properties of a certain vulnerability.

The computation is performed according to the formula given in [17] and the corresponding calculator [2]. Each of the three metric groups has its own characteristics and contains a set of metrics. Base metric group (Figure 1) describes the fundamental characteristics of vulnerabilities and is composed of the related exploit range, the attack complexity, the needed authentication level and the integrity, availability and confidentiality impact. Temporal metrics (Figure 2) are the metrics influenced by time passing, meaning the availability of exploit, the remediation level and the report of confidence. The environment (Figure 3) has also an impact when computing the score; environmental factors are the collateral damage potential, the target distribution and the confidentiality, integrity and availability requirement.

When talking about XSS exploits some of these metrics remain constant because of the type of this vulnerability. The related exploit range or the access vector is the network, because the attack is widely spread over the internet and the access complexity is medium. The majority of reports describe vulnerabilities on browsers having the standard, default configurations, but it is medium and not low, because the attacker is required to have some social engineering skills in order manipulate and fool custom users to access a certain page or click a specific button or link. In order to be considered successful, the attack has to gain information or control over the client machine and for this at least one instance of authentication is needed. The availability impact metric refers to the access of the attacker over the resources, meaning bandwidth, processor, disk space and his possibility to get a total shut-down of the affected resource. In case of an XSS exploit, we consider this availability impact to be zero; it is impossible from what we know until now for someone to access the resources using this type of vulnerability. We have intentionally skipped the confidentiality and integrity impact because they vary depending on the attack and we are focusing now on the constant metrics of XSS exploits.

[2]available at http://nvd.nist.gov/cvss.cfm?calculator&adv&version=2



Base Score Metrics

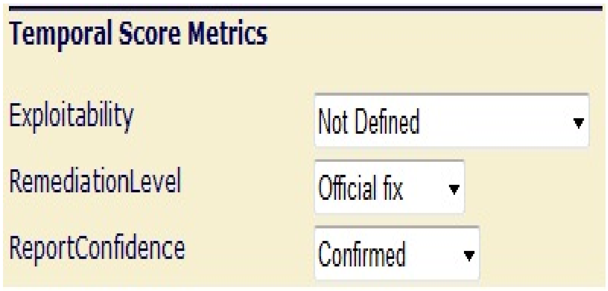
Exploitability Metrics

AccessVector	Network
AccessComplexity	Medium
Authentication	Single Instance

Impact Metrics

ConfImpact	Undefined
IntegImpact	Undefined
AvailImpact	None

FIGURE 1. Base scores metrics



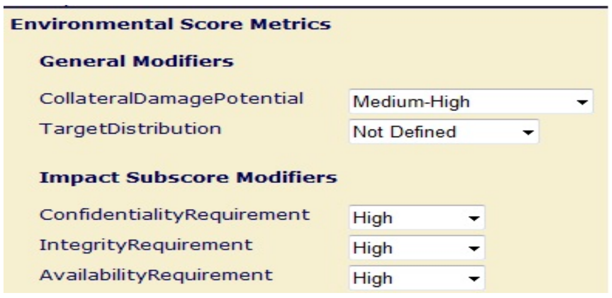
Temporal Score Metrics

Exploitability	Not Defined
RemediationLevel	Official fix
ReportConfidence	Confirmed

FIGURE 2. Temporal scores metrics

Moving to the temporal group, we will argue the chosen values to the metrics based on the vulnerabilities reported by Microsoft on their periodical security bulletins or by the other browsers in periodical advisories. We let the exploitability factor set to not defined, because officially they say that the exploitation code was not made public: ”Microsoft received information about this vulnerability through coordinated vulnerability disclosure.”, ”Microsoft had not received any information to indicate that this vulnerability had been publicly used to attack customers when this security bulletin was originally issued”, while the other browsers avoid to make this kind of statements naming only the person that reported the vulnerability. Moreover, all the vulnerabilities are confirmed and are reported only after an official fix is available.

The damage potential of an XSS vulnerability is according to OWASP moderate. We are not talking about a physical damage, but there can be significant loss of information. Last, but not least there are the impact requirement modifiers. Browsers are meant to be secure. Confidentiality, integrity and availability are the three features users require for safe browsing and financial transactions.



Environmental Score Metrics

General Modifiers

CollateralDamagePotential	Medium-High
TargetDistribution	Not Defined

Impact Subscore Modifiers

ConfidentialityRequirement	High
IntegrityRequirement	High
AvailabilityRequirement	High

FIGURE 3. Environmental scores metrics

The metrics that change depending on the XSS exploit are the confidentiality and integrity impact and the percentage of vulnerable systems. In order to see how these metrics differ we will consider four vulnerabilities.

URL Validation Vulnerability [18] is a critical vulnerability reported in February 2010 that appeared from incorrectly validated input. It provided the attacker access to the client machine with the same rights as the logged in user and if the attacker could reach administrative rights, he could install programs, read or change data. In this case, because remote code could be executed, the confidentiality and integrity impact is complete. Regarding the target distribution, which was Internet Explorer, it was reported as vulnerability on IE 7 and 8 which at that time covered 35.3 % of the market, so medium spread. The score in this case reaches 6.3.

Before the release of 10.63, Opera reported the existence of the following vulnerability: Reloads and redirects can allow spoofing and cross site scripting [23]. It allowed bypassing the same origin policy and execution of scripts in the wrong security context. One might also change configurations in the browser and gain access to the target computer. Opera, which then achieved a number of 2.1 %, described the vulnerability as critical, because the confidentiality and integrity impact was complete. Although the high impact, the flaw is rated with 2.1 because of the low spread.

ChromeHTML URI handler vulnerability is a vulnerability reported on Chrome on April 2009. It implied execution of arbitrary URI without any user interaction and could allow information leakage such as stealing of victim's cookies and directories and files disclosure, so confidentiality and integrity impact was just partial. It affected and was fixed on Chrome 1.0 targeted at that time with 4.9 % on the market. CVSS score in this case is 1.8.

March 13, 2012 brought to public's attention a vulnerability of the Content Security Policy implemented by Firefox. The vulnerability called XSS with multiple Content Security Policy headers [19] allows header injection into Firefox 11 (36.3 %). Because Firefox did not reported remote execution of code, confidentiality and integrity impact is considered partial and the score is around 5.5.

Target distribution is calculated taking into consideration data provided by http://www.w3schools.com/browsers/browsers_stats.asp, related to the date of the report.

The first remark is the importance of target distribution in calculation of score. If it slightly varies from low to medium the score increases with at least 2 points. The second observation is that the security policies adopted cannot face sophisticated attacks and can also provide security flaws.

These results can contribute to the evaluation of the business impact of XSS vulnerabilities. The browser-dependent risks must be carefully treated since they allow attackers to have end user privileges and to gain control of the applications. The computed scores confirm the OWASP evaluation and the position of cross site scripting vulnerabilities on their list [22].

6. Tool support

As you can probably imagine finding and testing security vulnerabilities is not an easy task. One has to be creative and determined in order to find all the entry points of an application and how they can bypass the validation methods added by the developer. Web Application Vulnerability Scanners are tools designed to automatically scan web applications for potential vulnerabilities. They perform a range of checks, such as field manipulation and input poisoning and enumerate the lacks found together with their target. Some of them also provide severity classification and general protection methods. Even if some claim differently, it seems that there is no security tool finding 100% of the vulnerabilities and avoiding false positive.

For testing purposes and in order to automate this process of finding web applications flaws and then try them on different browsers, we developed a security testing tool. *SecurityApp* is such an application. It is a software application designed for security vulnerabilities testing. It is a desktop solution that requires as input the URL of the desired web application and performs a set of tests in order to determine and confirm Cross-site scripting vulnerabilities. After running the tests, *SecurityApp* returns a report that displays the tests that failed, the application page, the parameter to which injection was performed and the injection string. It is an automated tool designed for testing, but it also allows manual checking. Users can add injection strings or can require testing of a specific page and parameters.

Briefly, the testing process follows the flow shown in Figure 4. The desktop application performs crawling base on the input URL. This means, it sends

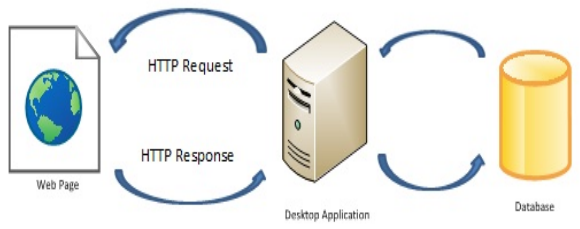


FIGURE 4. Testing process flow

several requests to the web application desired to be tested and analyzes the response by parsing the HTML code and looking for other accessible pages. After obtaining a new web page together with its GET and POST parameters, it initiates several attacks, injecting one of the parameters with strings retrieved from the database. The response is again parsed and analyzed in order to determine if it was successful or not.

Our work started by testing some of the available open-source or trials web security scanning tools in order to see the results on some web applications: ours or our colleagues and noticed that depending on the technology used the results were different, but also there was a huge difference between the performance, the vulnerability list and the initial configurations. One may be able to estimate the general capabilities of a scanner from the amount of REAL exposures that are located, the amount of exposures that are missing (false negatives) and from the amount of FALSE exposures (false positives) are identified as security exposures. The main focus is in not missing the present vulnerabilities so all the tested applications returned false positives or reported their occurrence as highly possible.

Because it addresses a certain type of vulnerability, *SecurityApp* performes an increased number of tests and tries to avoid the false positives by calculating a percentage of successful exploits for each parameter. The higher the percentage is, the more vulnerable that entry point is.

SecurityApp is a Windows forms application developed in .Net Framework 4. It performs crawling and cross-site scripting and it retrieves and stores data in a Microsoft SQL Server database. It consists of a set of modules used for different purposes: access to the database, models, user interface, crawling and XSS exploit.

Crawling is done based on the input given URL using the HTML Agility Pack library that allows reading and writing of DOM. It is an HTML parser supporting XPATH and XSLT for an easier interaction with the DOM objects.

Parsing is performed regardless the strict format of the HTML, so it is very tolerant to small errors in a malformed HTML response. Briefly, based on a string corresponding to an URL, it loads in the HtmlDocument the response received. The document is then used for manipulation. One might search for all DOM objects of some type or for all nodes having certain attributes. It may parse the children, check out for different values or add several nodes, attributes or events. XSLT is used for transforming XML documents and XPATH is used for addressing and referring parts of an XML document.

Considering these, we used HTML Agility Pack and XPATH to facilitate the parsing of the received response and collect all the referred links. We have checked for tags and forms. In case of a tag, we considered the *href* attribute and in case of forms the *action* attribute. If a form had no action attribute it meant the submit would be done to itself.

Before sending the request for the new page, we have checked that the page belonged to the domain. If for example, a web page had a link for Facebook authentication, Facebook application should not be tested. *SecurityApp* is meant for personal sites testing and not finding vulnerabilities over the internet.

Exploit operation for XSS is done using a web page and the injection scripts from the database. The application takes each parameter of the page and constructs a request for the page using an injection script and the parameters. Just one parameter is injected at a time in order to find the exact place of the page vulnerability. Parameters are sent via GET or POST and if not being exploited, they get the initial value or a default one. The strategy used is to construct scripts that can also bypass basic input sanitization:

- sanitize for apostrophe or for quotes, but not both
- avoid script tag ⟨SCRIPT⟩, but use ”javascript:” (specifier can also be used in HTML links)
- no use &, ⟨, ⟩ , # or ; characters
- no script tag ⟨SCRIPT⟩ or use of ”javascript” (successful when used in concatenations)
- encoded validation skip.

In order to confirm the vulnerability, the response page is analyzed. Each inserted script is expected to be found in the response, but in a certain place and having a certain format, if not the parameter must have been sanitized and the exploit was not successful.

The results are stored in the database for further processing in reports. The tool is in the early stages of development and have been design in order to offer an easy-to-use tool for testing the vulnerabilities for our own purposes.

Although several such tools exists, our application proved to be more efficent for our research purposes, due to its specificity.

7. CONCLUSIONS AND FUTURE WORK

The paper gives an overview of XSS vulnerabilities from a browser point of view. We studied the impact of URL Validation Vulnerability on Internet Explorer. Reloads and redirects can allow spoofing and cross site scripting on Opera, ChromeHTML URI handler vulnerability and XSS with multiple Content Security Policy headers vulnerability on Firefox. We have used the CVSS vulnerability scoring formula in order to measure the impact of these vulnerabilities on security and the obtained results confirm the OWASP analysis, for exploitability and impact.

It is our opinion that XSS vulnerabilities should be carefully treated and prevented. A combined protection approach involving writing secure code, proper testing and request validation made by the browser could eliminate most of the XSS attacks and improve significantly the security of each application.

As future direction of our study, we intend to analyze other forms of XSS vulnerabilities that are more difficult to perform and detect. We will continue testing ActiveX and Silverlight on Internet Explorer and Flash on the other browsers. We also intend to modify the default settings regarding security in each browser and observe the changes in behavior.

As part of our research we also develop a security testing tool that measures the vulnerability of an application to security attacks. This way we will automate the exploit mechanism and perform a larger number of tests.

References

[1] Acunetix, http://www.acunetix.com/

[2] J. Atwood, Coding horror, Protecting Your Cookies: HttpOnly, August 28, 2008

[3] Barracuda Networks, http://www.barracudanetworks.com/ns/products/web-site-firewall-overview.php

[4] BaydenSystems, http://www.bayden.com/tamperie/

[5] M. Brooks, Bypassing Internet Explorer's XSS Filter, Traps Of Gold-Defcon 2011, https://sitewat.ch/files/Bypassing%20Internet%20Explorer's%20XSS%20Filter.pdf

[6] CERT - Measuring Software Security Assurance - www.cert.org/research/2010research-report.pdf

[7] Cisco - ACE Web App Firewall http://www.cisco.com/en/US/products/ps9586/index.html

[8] Citrix NetScaler App Firewall http://www.citrix.com/English/ps2/products/product.asp?contentID=2312027

[9] G.A. Di Lucca , A.R. Fasolino, M. Mastoianni, P. Tramontana, Identifying cross site scripting vulnerabilities in Web applications, Proc. WSE 2004, pg. 71-80

[10] J. Grossman, S. Fogie, R. Hansen, XSS Attacks: Cross-site Scripting Exploits and Defense, Syngress, 2007

[11] M. Howard, D. LeBlanc, Writing Secure Code, Microsoft Press, 2003
[12] HP WebInspect, https://download.hpsmartupdate.com/webinspect/
[13] IBM Software, Reduce your cost of quality, http://www-01.ibm.com/software/rational/smb/quality/
[14] A. Klein, DOM Based Cross Site Scripting or XSS of the Third Kind. Web Application Security Consortium Articles, 4, 2005, http://www.webappsec.org/projects/articles/071105.shtml
[15] D. Lynch, XSS is fun!, October 20, 2011 http://davidlynch.org/blog/2011/10/xss-is-fun/
[16] Mavituna Security, http://www.mavitunasecurity.com/netsparker/
[17] P. Mell, K. Scarfone, S. Romanosky - A Complete Guide to the Common Vulnerability Scoring System Version 2.0, 2007, http://www.first.org/cvss/cvss-guide.pdf
[18] Microsoft - http://technet.microsoft.com/en-us/security/bulletin/MS10-002
[19] Mozilla Foundation Security Advisory, http://www.mozilla.org/security/announce/2012/mfsa2012-13.html
[20] MSDN - Security, Anti-Cross Site Scripting Library, http://msdn.microsoft.com/en-us/security/aa973814
[21] NASA - Open Source Summit 2011, http://www.nasa.gov/open/source/
[22] Open Web Application Security Project www.owasp.org
[23] Opera Support, http://www.opera.com/support/kb/view/973/
[24] PortSwingger Web Security, http://portswigger.net/burp/
[25] Social Hacking, Recent Facebook XSS Atacks Show Incresing Sophistication, April 21, 2011
[26] G. Wassermann, Static detection of cross-site scripting vulnerabilities, Proc. of ICSE 2008, pg.171-180

Babeş Bolyai University, Department of Computer Science, M. Kogălniceanu 1, 400084 Cluj-Napoca, Romania
E-mail address: `naie1000@scs.ubbcluj.ro, motogna@cs.ubbcluj.ro`