# A Multi–agent Scanner to Detect Stored–XSS Vulnerabilities

E. Galán, A. Alcaide, A. Orfila, J. Blasco
University Carlos III of Madrid, UC3M
Leganés, Spain
{edgalan,aalcaide,adiaz,jbalis}@inf.uc3m.es

## Abstract

*The cross-site scripting (XSS) has become a common vulnerability of many web sites and web applications. XSS consists in the exploitation of input validation flaws, with the purpose of injecting arbitrary script code which is later executed at the web browser of the victim. One interesting possibility to prevent this type of vulnerability is the use of vulnerability scanners. However, current scanners are capable of detecting just one of the two main modalities of XSS attacks. This paper introduces a novel multi–agent system for the automated scanning of web sites to detect the presence of XSS vulnerabilities exploitable by an* stored–XSS *attack. The rate of detection of the system is evaluated in two different scenarios.*

## 1 Introduction

The web applications provide users with a wide range of services, usually exhibiting high degrees of usability. Since people are frequently asked to enter private information into those applications to perform sensitive operations online (such as bank transactions), web applications have become a desirable target for cyber–criminals. In this regard, cross-site scripting (XSS) attacks on web applications have experienced an important rise in recent years [14, 13].

XSS exploits flaws in web applications which allow an attacker to execute arbitrary code without the authorization of the web application. This way, an unaware user can be the victim of an identity theft, electronic fraud or other modalities of cyber–crime.

XSS attacks occur in three main well differentiated ways: reflected–XSS, stored–XSS and DOM–based XSS. These modalities differ from each other in the way they manage to inject the intrusive code into the application and in the way this code is executed. The majority of authors do not include DOM–based XSS attacks when they enumerate the different XSS attack types [8]. The reason of that exclusion is not only the lower number of occurrences of that specific type of attack but also the different nature of the attack itself: while reflected and stored XSS attacks are due to vulnerable web applications, DOM–based attacks are motivated by vulnerabilities of the interpreter of the script used by the web browser.

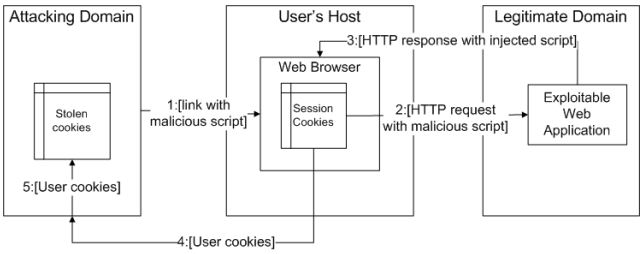


**Figure 1. Reflected–XSS attack**

Reflected and stored XSS attacks exploit vulnerabilities which are found on web applications. These attacks inject the script code through an HTTP request, usually as a parameter or input of a web form. In reflected attacks, the injected script is immediately executed in the browser of the victim as the script is included in the response to the HTTP request. By contrast, stored–XSS attacks work in a different way: their goal is to inject the script in a persistent way. This way, an attacker has to exploit a vulnerability just once and the injected script would execute as many times as the web page containing the script is visited. Figures 1 and 2 show the diagrams of examples of reflected–XSS and stored–XSS attacks respectively.

Among the multiple possibilities for mitigating the impact of XSS attacks, vulnerability scanners have proven to be valid tools for that task. However, their range of coverage is limited as these tools just take into account reflected–XSS attacks. It is the purpose of this paper to expand the coverage of vulnerability scanners to include stored–XSS attacks too. This way, vulnerability scanners can become a complete and integral solution to reduce the presence of XSS vulnerabilities on web applications.

 

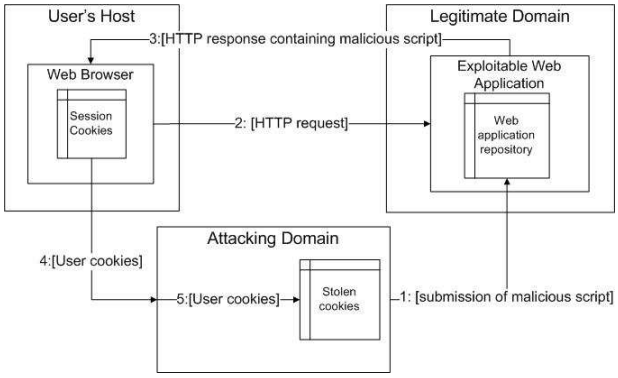


**Figure 2. Stored–XSS attack**

## 2 Literature Review

XSS is a security threat which has been addressed by researchers throughout a variety of approaches. Existing proposals range from design methodologies, applicable by web developers, in order to prevent XSS attacks, to detection tools that identify XSS vulnerabilities on operating websites.

Though the ideal solution would be the use of design methodologies that take into account the security aspects of a web application [11], those aspects are very frequently overlooked. This implies that a high number of XSS vulnerabilities are not detected during the development phase and can be exploited later on by potential attackers.

To overcome this problem, different tools are usually deployed in a cumulative way over several security layers. Some authors have proposed the use of static analysis techniques to discover input validation flaws in a web application, however, this approach requires access to the source code of the application [16, 5]. Moreover, those static analysis schemas are usually complemented by the use of dynamic analysis techniques [3, 15, 2]. The dynamic analysis is used to confirm potential vulnerabilities detected during the static analysis by watching the behavior of the application at runtime.

Several existing systems have been adapted to detect XSS. Application level firewalls [7], reversal proxies [17] and IDS (Intrusion detection systems) ([10, 4]), have been adapted to try to mitigate the XSS problem. Firewalls focus on tracking sensitive information and controlling whenever data is to be sent to untrusted domains. Reverse proxies receive all responses from the web application and check whether there are any unauthorized scripts on them. IDS approaches deal with the identification of traffic patterns that allow the detection of known XSS attacks.

The use of vulnerability scanners [6, 1] facilitates, to a great extent, the automated search of XSS vulnerabilities in web applications. They evaluate the application by launching a collection of attacks against the application and verifying if they were or not successful. That verification process simply consists in checking if the injected script is present in the response obtained from the application, so this approach is only capable of detecting vulnerabilities that can be exploited through reflected–XSS attacks but not stored–XSS attacks.

It is the goal of this work to complete the scope of vulnerability scanners by allowing them to check the presence of stored–XSS vulnerabilities in web applications.

The system we proposed is based on a multi–agent architecture with the following main characteristics:

1. The multi–agent architecture allows the different agents to be able to operate independently, so the system is more efficient than those based on a single operator.

2. The vulnerability scanner does not need to have access to the source code of the scanned application.

3. The architecture is highly flexible and adaptable for the scanning of any web application.

4. The proposed system can be used by developers and web managers to enhance web security.

5. It can be used by external entities in web security auditory processes.

6. Acting in coordination with other existing reflected–XSS vulnerability scanners it offers a complete coverage against XSS attacks.

The remainder of this paper is organised as follows. Section II explains our multi-agent proposed system. Section III shows the evaluation methodology applied and the results obtained. Section IV gathers our conclusions and finally, Section V enumerates our future work directions.

## 3 Multi–agent architecture

Present work focuses on the extension of current XSS vulnerability scanners with the purpose of making them able of scanning applications looking for stored–XSS vulnerabilities. As current vulnerability scanners are only capable of detecting reflected–XSS vulnerabilities, the capability of identifying all types of XSS flaws would constitute an important step towards turning this kind of scanners into an efficient and integral tool for preventing XSS. Regular XSS vulnerability scanners (see Figure 4 for a graphical representation) work in the following way :

 

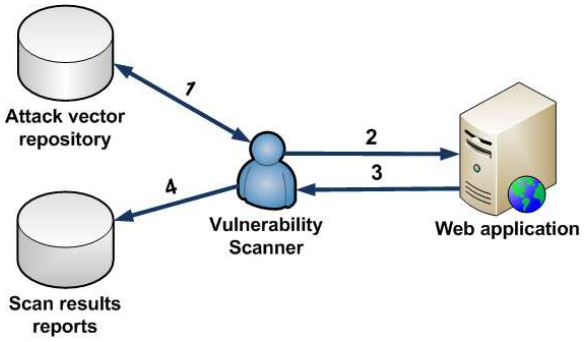


**Figure 3. Reflected–XSS vulnerability scanner**

1. A selection of attack vectors are obtained from an attack vector repository[1].

2. Selected attack vectors are launched against inputs of the web application. Those attack vectors are generally injected in a HTTP request as parameters or as fields in a web form.

3. The vulnerability scanner receives the responses to the requests which contained the injected code.

4. The vulnerability scanner checks for the presence of injected script in the received responses. If affirmative, XSS attack is considered successful and a vulnerability of the scanned web application has been discovered.

Note that scanning an application looking for reflected–XSS vulnerabilities is a trivial task, as looking for the injected script in the immediate response can be carried out in a straightforward way. By contrast, to scan for stored–XSS flaws is a more complex task. This fact is due to the greater difficulty of verifying whether the script was successfully injected or it was detected and properly processed by the web application. As the injected script is not immediately returned to the web browser but stored among the legitimate content of the web application, a tool that scans for stored–XSS vulnerabilities must be able to locate the injected script among the rest of content. To sum up, our system must accomplish the following objectives:

- Finding the input points of the application susceptible of being vulnerable to a stored–XSS attack.

- Injecting selected attack vectors at the previously detected points.

- Crawl through the web application looking for the injected scripts in order to verify the success of the attack.

Our novel multi–agent architecture allows for each one of those tasks to be carried out by a different type of agent. This design decision has been taken to allow each of the stages of the scanning process to be performed concurrently with the other stages. It also allows for the different subtasks of the scanning process to take place in a distributed and/or parallel way. The set of agents part of the proposed architecture and the operation of the scanner are listed below (also see Figure 4):

1. A **webpage parser agent** crawls the web application.

2. Information about the different web forms found in the previous step is used to build a repository of potential injection points (*Injection point repository*).

3. A **script injector agent** reads the list of injection points identified by the parser agent.

4. The script injector agent also makes a selection of vectors attacks from the *Attack vector repository*.

5. The desired set of attack vectors is launched against each of the potential points of attack of the application.

6. A list of the performed attacks is stored in a *Performed attack list*.

7. The **verificator agent** gets the list of the attacks to be verified.

8. The verificator agent crawls the web application looking for each of the attacks.

9. A report about the results of the scanning process is elaborated and stored.

A more detailed description of each agent is given in the following sections.

### 3.1 Web Page Parser Agent

It is an agent that explores the web site in order to find the injection points where stored–XSS attacks could be launched. This parsing process is similar to that of web crawlers and spiders [9], as it systematically retrieves information from the pages it visits and it propagates through the site following the hyper-links it finds. Nevertheless, it differs from the typical web crawler in two aspects: (1) It just follows the hyper-links with destination to the scanned site discarding all external links and, (2) The information recovered are web forms. Web forms have been chosen as the point of entry for stored–XSS attacks due to the fact that

[1]XSS attack vectors are commonly stored in repositories and include the description of the attack as well as the script code to be injected.

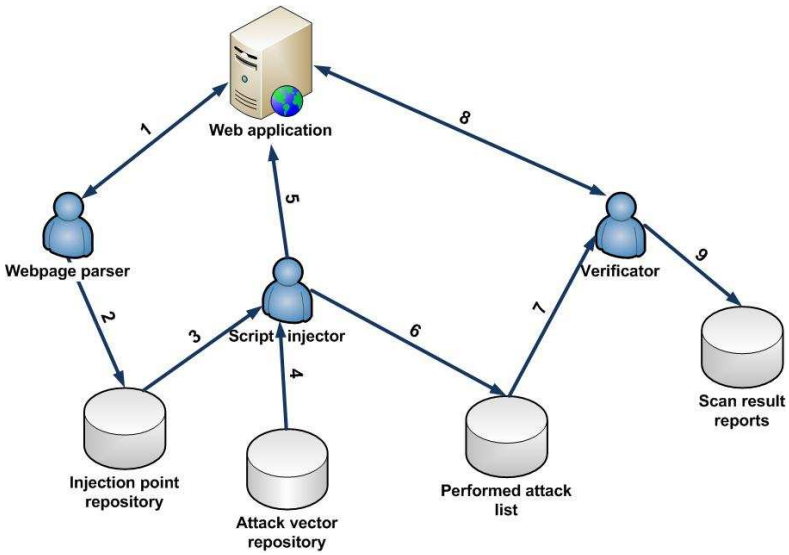


**Figure 4. Multi-agent architecture of a scanner to detect stored–XSS vulnerabilities**

| Attribute | Description | |
|---|---|---|
| Name | The attribute *name* of the HTML form. | |
| Action | The *action* field indicates the destination of the form data. | |
| Method | The method of the HTTP request originated when the form is submitted. | |
| | Control | The type of control of the field: input, textarea... |
| | Type | Specific type of the control: password field, submit button, text field, radio button... |
| | Name | The *name* attribute of the field |
| | Value | The value of the field. |

**Table 1. Injection point repository entry**

they are the main mechanism offered by web applications to add new content that is persistently stored. Additionally, web forms that sends their data to a different website are not taken into account either. Links with parameters are not stored as entry points as they are usually used to query the database used by the web application, and not to insert new information, (they can be used to launch reflected–XSS attacks, but not stored–XSS attacks).

The output of this agent is a collection of forms which are likely to be vulnerable and potential targets for stored–XSS attacks. As previously stated, such information is used to build an Injection point repository. Table 1 illustrates an entry in that repository.

### 3.2 Script Injector Agent

This agent makes use of the collection of web forms elaborated by the web page parser agent and registered in the Injection point repository. The agent will inject a collection of XSS attack vectors from a well-known repository [12] into the different input fields of each of the injection points.

The set of attacks used for the evaluation of our tool were extracted from a repository of XSS attack vectors in [12]. Those vectors make use of different ways of inserting arbitrary script code trying to be unnoticed by the web application and, in our case, to be incorporated as legitimate content in the web application. Attack vectors in the repository are widely varied and they are classified as follows:

- Basic XSS vectors: direct injection of the malicious script.
- HTML Element vectors: malicious script is injected along with regular HTML elements.
- Character Encoding vectors: different ways of representing text are used to get the script injected.
- Embedded Character vectors: consists in the insertion of deliberately incorrect characters among the script content with the purpose of making the script unnoticeable by the web application but executable by the web browser.
- Event Handlers vectors: they try to inject the scripts as JavaScript event handlers such as *onClick*, *onLoad*, etc.
- XSS with HTML Quote Encapsulation: they actively try to evade input validation filters, specially the ones consisting of regular expressions.
- URL Obfuscation vectors: they focus on modifying an URL in order to make it unrecognizable by the input validation filters of the web application.

 

| Attribute | Description |
|---|---|
| Name | Identifies the attack vector. |
| Code | The malicious code which will be injected. |
| Description | A brief explanation on what the attack consists in. |
| Label | The category in which the attack vector is contained. |
| Browser | The specific version of web browsers where the injected script can be executed. |

**Table 2. Attack vector repository entry**

| Attribute | Description |
|---|---|
| Page | URL where the script code was found. |
| Code | Script code found corresponding with one of the launched attacks. |

**Table 3. Attack vector attributes**

| Attack type | Launched attacks | Detected attacks | Detection rate |
|---|---|---|---|
| Basic XSS Attacks | 5 | 2 | 40% |
| HTML Element Attacks | 47 | 12 | 25.53% |
| Character Encoding Attacks | 13 | 6 | 46.15% |
| Embedded Character Attacks | 17 | 6 | 35.3% |
| Event Handlers | 0 | 0 | 0% |
| XSS with HTML Quote Encapsulation | 7 | 4 | 57.14% |
| URL Obfuscation | 14 | 12 | 85.71% |
| Other Attacks | 7 | 2 | 28.57% |
| **Average detection rate:** | 39.8% | | |

**Table 4. Vulnerable web site results**

- Other attacks: singular attack vectors which are not classifiable in any of the previous categories.

Each of the attacks contained in the Attack vector repository is characterized by a series of attributes described on Table 2.

The script injector agent registers the different attacks it launches into the *Performed attack list*.

### 3.3 Verificator Agent

The third and last agent of the proposed system takes as input the list of performed attacks, which was produced by the script injector agent, and looks for those attacks in the analyzed web application. It is necessary to search for the injected scripts because, in opposition to reflected–XSS attacks, the success of stored–XSS attacks can not be verified immediately in the response to the HTTP request which contained the malicious script code.

After this second crawling of the web site it is possible to determine if the web application shows any of the stored–XSS vulnerabilities identified during the scanning process.

The results obtained are very useful in the process of securing the application by correcting the input validation errors which led to the success of the attacks. The results of the scan are structured as it is shown in Table 3.

## 4 Evaluation

An implementation of the proposed system was developed with the purpose of testing and evaluating the scanner against different web sites. The implementation of our multi-agent system allows the three different agents to operate separately. However, a sequential execution is motivated by the strong dependency between the agents as the input of an agent is the output generated by the previous one. This way, the first agent to be launched would be the Webpage parser agent. Next agent is the Script injector agent, which uses as input the attack point repository generated by the previous agent. Finally, the Verificator agent which cannot be executed until the list of launched attacks is generated by the injector agent.

Our experimental work focused on two different scenarios. The first series of experiments carried out were against an unsecured application. The scanned web application is a highly-simplified web for a social network. Users can post text entries and those entries can be commented by other users as well as by themselves. The web application has been implemented in the Java language on an Apache Tomcat servlet container. The application data is stored in a MySQL database. The application does not perform any validation on the received input so a high number of XSS vulnerabilities are expected.

The use of this first experimental framework served to test and evaluate the usability of each of the agents, to integrate the different parts of the architecture and to measure the effectiveness of the whole system.

The second series of experimental work was against a web application which had been developed by a freely distributed content management tool such as Drupal. Drupal is a popular content management system which allows the easy creation and management of web sites. This experimental setting allowed us to validate the tool against real scenarios.

The results obtained differ greatly on each of the used scenarios. In the first case many attacks were detected by our scanner (Table 4) reaching an average detection rate of $39.8\%$. Unsuccessful attacks were due to two main reasons: (1) The attack vector repository, though it is a comprehensive and wide collection of attack vectors, is more focused on showing the different possibilities of XSS attacks rather than providing ready–to–use attack vectors and, (2) Many of the launched attacks were thwarted by the input validation mechanisms of MySQL.

By contrast, the evaluation of the Drupal–based web site did not show any successful attacks. This was an expected result, as the evaluated web application was framed within a consolidated and widely-tested software.

 

## 5 Conclusions

Vulnerability scanners are a promising mechanism to fight the presence of XSS vulnerabilities in web applications. Current proposals allow to automatically look for that kind of security holes, although they also present an important limitation: they cannot detect stored–XSS attacks.

Finally, an implementation of the scanning system was tested against two different scenarios. The result attained in both series of experiments served to guarantee the correctness and efficiency of the proposed scheme.

## 6 Future Work

The proposed system constitutes a complete working architecture, however, some aspects can be modified in order to improve the scanning process by getting a better performance and accuracy.

The selection of the points of the webpage where input data is incorporated to the content of the web application can be optimized by the use of certain rules and heuristics which would represent an important performance boost as code injection would be performed at input points where a stored–XSS attack is more likely to be successful.

The repository [12] used for evaluating this tool contains a representative collection of attack vectors which illustrates the different possibilities of XSS attacks. However, that implies that its attacks are simple enough to be detected by basic validation input mechanisms and are not well-suited for circumventing additional input filters such as the validation performed by the database engine. Defining a more effective collection of attack vectors would imply greater possibilities of obtaining better results (more vulnerabilities identified) from a scanning process.

## 7 Acknowledgement

This work has been partially supported by CDTI (Ministerio de Industria, Turismo y Comercio of Spain) in collaboration with Telefónica I+D, Project SEGUR@ with reference CENIT-2007 2004

## References

[1] Acunetix. Acunetix, web application security, 2010.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy, 2008. SP 2008*, pages 387–401, 2008.

[3] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM New York, NY, USA, 2004.

[4] M. Johns, B. Engelmann, and J. Posegga. Xssds: Server-side detection of cross-site scripting attacks. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 335–344. IEEE Computer Society Washington, DC, USA, 2008.

[5] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy*, page 6, 2006.

[6] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, pages 247–256. ACM New York, NY, USA, 2006.

[7] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337. ACM New York, NY, USA, 2006.

[8] A. Klein. Dom based cross site scripting or xss of the third kind, 2007.

[9] M. Kobayashi and K. Takeda. Information retrieval on the web. *ACM Computing Surveys (CSUR)*, 32(2):144–173, 2000.

[10] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 251–261. ACM New York, NY, USA, 2003.

[11] OWASP. Xss (cross site scripting) prevention cheat sheet, 2007.

[12] RSnake. Xss (cross site scripting) cheat sheet, 2009.

[13] SecurityFocus. Bugtraq mailing lists, 2009.

[14] A. Stock, J. Williams, and D. Wichers. Owasp top 10. *OWASP Foundation, July*, 2007.

[15] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS07)*, 2007.

[16] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th international conference on Software engineering*, pages 171–180. ACM New York, NY, USA, 2008.

[17] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel. Swap: Mitigating xss attacks using a reverse proxy. In *Proceedings of the ICSE Workshop on Software Engineering for Secure Systems (SESS '09)*, 2009.

 