Detecting vulnerabilities in Web applications by clustering Web pages

V. Suhina, S. Groš, Z. Kalafatić
Department of Electronics, Microelectronics, Computer and Intelligent Systems
Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, 10000 Zagreb, Croatia
Phone: (01) 612-99-35 Fax: (01) 612-96-53 E-mail: vanja.suhina@fer.hr

Abstract – In this paper, we propose a new approach to detecting vulnerabilities in Web applications. Majority of current Web application vulnerability scanners rely on detecting vulnerabilities by detecting common error messages or input vectors used in testing. The method we propose in this paper is based on detecting unusual behavior of a Web application. Differences between pages are analyzed by examining page structure, i.e. HTML elements. Variations from standard page structure could indicate raised errors in the Web application and could indicate a vulnerability. Issues that arise in building such a tool will be described here.

I. INTRODUCTION

Most Web applications are susceptible to some kind of vulnerability. Last year's research shows that over 80% of Web applications were vulnerable to Cross Site Scripting (XSS) and over 25% were vulnerable to SQL injection flaws [1]. These are also the vulnerabilities that are most often searched by Web application vulnerability scanners.

Trying to find all the vulnerabilities in a Web application is a complicated and time-consuming task, so available scanners often concentrate on vulnerabilities that are easier to detect and that are more widespread then the others.

After analyzing the source code of some popular open source Web application vulnerability scanners, we found out that they all work in a similar way. They all detect vulnerabilities by finding common error messages or finding input values echoed back to the user. If the Web application uses custom error messages without additional unnecessary information for the user, these methods of detection fail.

On the other hand, some of the vulnerabilities are very difficult or even impossible to detect by automated scanners so human intervention is needed. However, testing without automated scanners is highly impractical because of large amount of input parameters that some Web applications have and the large amount of input values that should be used for each parameter. The solution is to combine automated scanners and human intervention.

This is the reason why we also took the same approach. We used the computer's capability to process a large amount of data, collecting various pages with big sets of input parameters and input values. User's intervention is needed to configure parameters and values for testing and to decide which of the results are indications of vulnerabilities.

In this paper the following terms will be used. When talking about Web page's non-HTML content, i.e. text that can be seen on the screen, it will be simply referenced as content. When talking about Web page's HTML content, i.e. HTML elements, it will be references as Web page's structure.

The paper is structured as follows. In the second section we present the list of the most critical Web application vulnerabilities. We also describe how the Web application vulnerability scanners try to detect those vulnerabilities. In the third section, the new method for detecting vulnerabilities is proposed and explained. The issues that arise in building such tool will be described in the fourth section. The paper finishes with conclusion and list of references.

II. OVERVIEW OF THE EXISTING VULNERABILITY DETECTION SYSTEMS

In this section we describe how the current Web application vulnerability scanners detect vulnerabilities. Detecting vulnerabilities is generally not an easy task, and not all of the common vulnerabilities can be successfully detected by automated scanners.

The list of top 10 most critical vulnerabilities, according to [2], is shown in Table I.

TABLE I
TOP 10 VULNERABILITY LIST

A1.	Cross Site Scripting (XSS)
A2.	Injection Flaws
A3.	Malicious File Execution
A4.	Insecure Direct Object Reference
A5.	Cross Site Request Forgery (CSRF)
A6.	Information Leakage and Improper Error Handing
A7.	Broken Authentication and Session Management
A8.	Insecure Cryptographic Storage
A9.	Insecure Communications
A10.	Failure to Restrict URL Access

Cross Site Scripting and Injection Flaws are the two most spread vulnerabilities as noted before. These are also the two that are most easily detected by scanners. That is why most of the available scanners search for them, more or less successfully. The most of vulnerability scanners look only for particular subsets of these vulnerabilities: SQL injection is a typical injection flaw that is most often searched for, and reflected XSS is the easiest XSS type to detect.

While searching for a XSS vulnerabilities, most of the popular open source tools, such as Web Application Attack and Audit Framework (w3af) [3] or Wapiti [4], work in a similar way. The tool requests a page from Web application

and provides some test payload that is inserted in one of the parameters that is being checked. If the payload is found immediately in the response page or somewhere later in the application it is a sign of a vulnerability. If the data is validated and sanitized, the Web application doesn't echo back the test payload. In these cases it usually returns default page, blank page or error page.

Searching for all kinds of injection flaws (i.e. SQL, XPath, LDAP, MX, etc.) is done in a similar way. Scanners again provide testing vectors through various input parameters and then in the response page they try to find some common error messages or part of it as an indication that it managed to raise an error in the Web application. If the Web application is configured in such a way that it doesn't echo back system error messages, these methods fail. There is also a method called blind injection [5] [6] that is able to detect and exploit injection flaws in Web applications when no error message is echoed back.

Other vulnerabilities shown in Table I. are not as easily detectable. It is because of scanner's inability to understand the data or know how the Web application is supposed to work. For example, the scanner cannot know for sure if some parameter references some internal object, and if the user is supposed to see altered values, i.e. other objects. In the same way, the scanners cannot know if some Universal Resource Locator (URL) should be restricted and guarded by authentication and authorization mechanisms.

Each found XSS vulnerability also implies that there exists CSRF vulnerability, but finding stand-alone CSRF vulnerabilities is a difficult problem.

Scanners are good at detecting information leakage and improper error handling and this is being done also by searching for some common strings that could represent some forgotten data in comments or commented code, or simply by displaying information on the pages (most often, error pages).

Detecting the use of insecure protocols for transmitting sensitive information is a trivial task for all the scanners, but they cannot know how this information is transmitted between backend servers. Also, the scanner cannot know in any way how this information is stored, if there is some encryption in use and if it is configured properly.

Detecting broken authentication and session management is also difficult for automated scanners as they cannot know which parts of Web application should be protected by access control mechanisms. Scanners also have trouble authenticating to the system and need human intervention for that part. However, they are useful for analyzing session identifiers for relative predictability.

III. VULNERABILITY DETECTION BASED ON CLUSTERING WEB PAGES

As shown in the previous section, two most common types of vulnerabilities can be found by probing the Web application with various input vectors and examining the response pages.

We propose a new method for detection not based on analyzing page's content. Our method analyzes response page's structure and finds deviations from standard structure. Differences are analyzed in responses from a single page or form requested with various input vectors.

First, various input values must be constructed for use in testing. Input vectors should cover different values, both meaningful and harmful. Input values should be linked with input parameters and form input vectors. The number of values for a single parameter and the rate between meaningful and harmful values should be decided in further research.

Gathering all the response pages retrieved with various values for input parameters is the next phase. The response pages have to be analyzed and grouped to several categories with respect to their structure. We suppose that all the valid pages will have similar structure and should be grouped together. When some error occurs in the Web application, the response page may have significant variations in structure and should be grouped to another category.

Finally, the representative pages of the groups should be displayed to the user so he can decide which of the categories represent odd behaving of the Web application. At that point, the user can see which of the input vectors produced particular behaving and focus the further search for vulnerabilities.

Various approaches can be chosen for grouping Web pages based on different measures of similarity. The majority of the current methods [7][8][9] use supervised learning for this purpose, i.e. classifying. These tools first have to build the classifier using the test examples of pages and then use this knowledge to differentiate new pages.

It would be impractical for user to browse through each Web application and find out what is the common structure of that particular Web application. Thus, supervised learning is not appropriate for this task and the decision was made that our tool should use clustering. Furthermore, it should run without any knowledge of the Web pages' structure.

There are already methods for clustering Web pages [10] but they are all based on web page's content. There are some that also take into consideration the web page's structure [11], but they still cluster pages regarding the content.

All of the mentioned methods use occurrences of some common words or phrases related to category and group the page to category according to found content.

We wanted to differentiate Web pages based on their HTML content, i.e. HTML elements. The differences should be examined regarding how they seem to a human user without reading the content. If this can be built successfully, the deviations in a page's structure should be detected and misbehaving of the Web application could be isolated in a separate group. The representatives of groups will then be displayed to the user and the user will decide which group represents the misbehaving. The user's response could then be used to refine the clustering criteria.

IV. INITIAL RESULTS

In this section, we will propose a design for a tool for vulnerability detection described previously. The architecture should be modular in order to allow the user to easily replace particular modules. We propose the architecture with four different modules as shown in Figure 1. and three intermediate file formats that connect the modules. In that way, the result files from one module can be used in some other way if necessary, or the modules could be replaced. The first module named *crawler* should be designed for examining the Web application and finding all the pages and forms and their input parameters. The information gathered in this module should be stored to an intermediate file. The human intervention is optionally needed before the next step. We want to provide the user an opportunity to select parameters and input values that he wants to use in a test. This will be handled by fuzzing options.

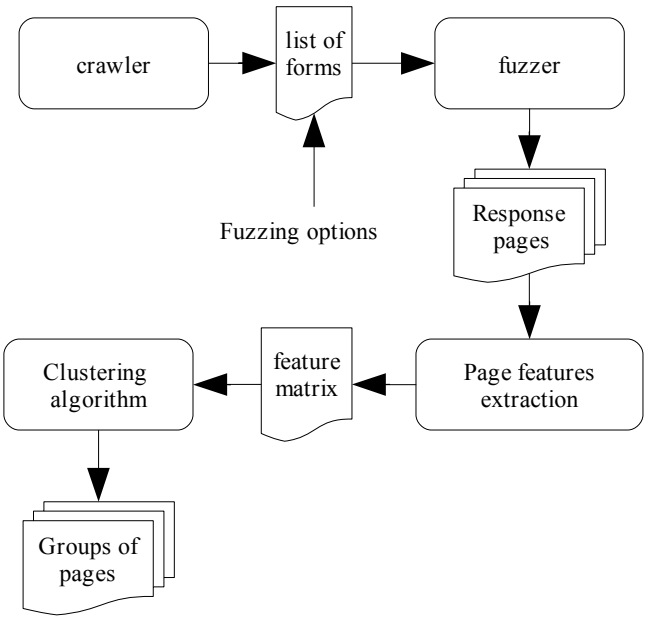

Fig. 1. Architecture design

The module named *fuzzer* should take the instructions from the user, generate input vectors and gather all the response pages. Beside the HTML content of the response page, these should contain information how was this particular page obtained and what input values for parameters were used. These pages should then be handed over to a module for *page features extraction* which will prepare the data for clustering. The document that contains vectors that describe each page is called feature *matrix* and should be designed in a way that can be used with various algorithms. *Clustering algorithm* module takes this document, performs grouping and as a result gives us pages that are good representatives of the clustered groups.

The problems with crawling the Web and gathering all the information about input parameters are already well known and will not be described here.

However, there are several issues associated with page features extraction and clustering algorithms:

1. Page structure representation
2. Number of groups
3. Group representatives
4. Input vectors

A. Page structure representation

The structure of the Web page should be transformed to multidimensional space containing page's features presented in numerical values. The main problem in building a tool for clustering based on the ideas presented in the previous section is a representation of the page's structure.

The deviations between response pages should be searched for a single web page of an application at a time, varying only one or several of its parameters. While experimenting on the Internet to find out how a change in a single parameter can result in a page's appearance variation we found two extreme conditions. In the first case, changing of a single parameter value can result in a minor change on the page, possibly only text information regarding the object that is described and referenced by the parameter. In other extreme condition, Web application can consist of a single page with one parameter and varying the value of the parameter can echo back to user the home page, article page, contact page or any other page in the application.

Our clustering algorithm should deal well with both extreme conditions found on the Internet. In the first case, it should detect even small deviations from the original structure, e.g. error has occurred in some module and it's displayed in a small portion of the page. In the latter case, it should form a global picture of the common Web application's structure and detect deviations from that structure.

B. Number of groups

The second problem which we faced is the number of groups used for clustering. As we don't have any a-priori knowledge on Web application pages' structure we don't know how many groups for clustering do we need. Even if we knew that varying the parameter's value results in a similar structure, we cannot know if there is one or more different errors and associated error pages that can be raised and displayed to the user. Selecting an algorithm which can dynamically decide number of groups is a subject of a further research. If the algorithm needs to have a predefined number of groups, it can be run with various number of groups and then, the results can be compared to find out which set of groups gives the most compact group. In this way, it is possible to simulate dynamic change of number of groups. It is also possible that results for various group numbers can give us additional information about the samples. For this purpose, the K-means algorithm should be appropriate.

C. Group representatives

The first goal of this approach was to automatically filter pages that represent odd behaving of Web application and display to the user these pages and input vectors that caused them. However, during the research, we found out

that the tool cannot know what is the normal behavior of the Web application and what is misbehaving. As a result, it leaves us with an option to reveal to the user all the group representatives so he can decide what is normal behavior, and what is wrong. When selecting pages that should represent the groups, it was decided that medoids of the groups are most appropriate. Medoid is a data sample that is closest to the center of the group.

D. Input vectors

Once the mechanism for grouping different pages is selected, there are still some areas that require experimenting. For grouping pages we use a series of response pages retrieved with various input vectors. It is a subject of further research to determine the optimal range of the group of input values, and at what rate should it contain meaningful values (i.e. values that are common in Web application and should return valid page) and harmful values (i.e. values that are supposed to raise errors). It is possible that clustering results would be better if there is only one or few valid pages and the rest are error pages, but it can also be possible that we would need more valid pages to be able to distinguish them from error pages.

V. CONCLUSION

Testing Web application vulnerabilities has become very important. It is known fact that detection of vulnerabilities can not be done solely by automated tools. It is also obvious that the tools speed up the detection process. The combination of automated tools and human perception is a way to efficiently search for vulnerabilities.

The method for detecting vulnerabilities proposed in this paper deals with this two-step process of assessing Web applications' security.

It was proposed how this tool should be built and what are the problems that arise in building and using such tool.

We believe that it is possible to find appropriate attribute space for the clustering mechanism that will be able to successfully distinguish different pages regarding only their structure, and not their content.

Experimenting with various Web applications should show if this approach speeds up the process of vulnerability detection and if achieves some new valuable information for the security researcher or penetration tester.

Acknowledgement

This work has been carried out with project 036-0361994-1995 Universal Middleware Platform for Intelligent e-Learning Systems funded by the Ministry of Science and Technology of the Republic Croatia.

REFERENCES

[1] WASC Web Application Security Statistics Project, http://www.webappsec.org/projects/statistics/

[2] OWASP Foundation, The Ten Most Critical Web Application Security Vulnerabilities

[3] w3af - Web Application Attack and Audit Framework, http://w3af.sourceforge.net/

[4] Wapiti - Web application vulnerability scanner / security auditor, http://wapiti.sourceforge.net/

[5] SPI Labs, Blind SQL Injection

[6] A. Klein, Blind XPath Injection

[7] D. Shen, Z. Chen, Q. Yang, H.-J. Zeng, B. Zhang, Y. Luand W.-Y. Ma, Web-page Classification through Summarization, Proceedings of SIGIR ‘04, 2004.

[8] S. Dumais, H. Chen, Hierarchical Classification of Web Content, Proceedings of SIGIR '00, 2000.

[9] X. Peng, B. Choi, Automatic web page classification in a dynamic and hierarchical way, Proceedings of the IEEE International Conference on Data Mining (ICDM’02), 2002.

[10] D. Boley, M. Gini, R. Gross, E.H. Han, K. Hastings, G. Karypis, V. Kumar, B. Mobasher, and J. Moore, Partitioning-Based Clustering for Web Document Categorization, 1999.

[11] E. J. Glover,K.Tsioutsiouliklis, Using Web Structure for Classifying and Describing Web Pages