International Journal of Innovative Technology and Exploring Engineering (IJITEE)
ISSN: 2278-3075 (Online), Volume-8 Issue-5, March 2019

# Developing a methodology for web applications vulnerabilities analysis and detection

**Mohamad Ahmad Swead, Muhammad Mazen Almustafa**

***Abstract*: *Background. Recently, web applications have proliferated rapidly, with the world increasingly dependent on financial transactions, purchasing, billing, education, medicine, and many more. But the security of these applications is worrying where any vulnerability might lead to breaches causing huge damages. In order to increase the security of these applications towards injection attacks, developers have followed a series of procedures, one of them is encrypting parameters and data before sending it. Decrypting these parameters gives attackers opportunities to target web application by launching injection attacks. We have developed a methodology to detect injection vulnerabilities by trying to decrypt hidden parameters which encrypted using MD5, SHA1, SHA2, AES. Methods. In order to implement the proposed methodology, a scanner called DEHP has been developed, DEHP employ a black-box approach to analyze targeted web applications. DEHP is using traditional crawlers to crawl and collect all URLs included in that application and for each single URL does the following steps: analyzing HTML syntax, extracting input parameters which its type is hidden, checking if these parameters' value is encrypted or not, if it's encrypted DEHP launching one of two types of attacks, dictionary or brute-force attack depending on user selection to try decrypt of hidden parameter value, DEHP is also checking Cross-Site scripting vulnerabilities by analyzing JavaScript syntax looking for commands related to interaction with the database, checking SQL injection blind vulnerabilities by launching attack towards input nodes (Username, Password), checking HTTP header type and date and finally checking digital certificate of HTTPS connections to make sure of its validity. DEHP has been developed under Visual Studio 2017 environment using C# and ASP .NET framework. Results. DEHP has been tested towards many web applications taking into consideration laws governing for such applications. Crawling speed was very good due to use traditional crawlers, detecting vulnerabilities speed was good using a dictionary attack (the database needs to be extended), by using brute-force attack speed was bad due to miss a suitable test bed and resources for such type of attacks. Results of DEHP were compared with similar open-source applications but none of them care about decryption of encrypted hidden parameters.*

***Index Terms*: *Vulnerability, SQL injection, Encryption, Crawler, Threats, XSS, Assessment tools.***

## I. INTRODUCTION

A common misconception about information security is related only to a computer! However, information security relates to all aspects of access, processing, transfer and storage information, whether through electronic or paper means. From the very beginning of the advent of technology and computers and throughout their growth period, information security has been a difficult challenge [1]. Security vulnerabilities in web applications are caused by many reasons such as human errors, errors in functional analysis of application, stop support and technical development of applications which make it vulnerable to various types of attacks, adding new functions to application by developers which requires writing many codes that increase the probability of coding errors and thus increase the chances of security vulnerabilities and one last important reason is neglect of developers to security aspects in web applications and lack of sufficient knowledge and training in this regard [2]. Based on web security indicators (WSI), through which we measure the security of the web application based on many indicators that affect the security of the application such as application settings, historical data related vulnerabilities, etc., we can see in below diagram how less than 5% of companies were protected and 40% still needs many preventative measures in order to be protected [3].

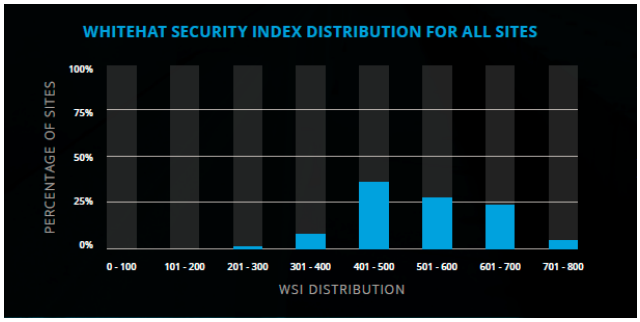


Figure 1. WSI Indicator for web applications security

Growing in vulnerabilities which lead to data breaches will cost companies and organization around the world about 3 trillion dollars annually from 2015 and expected to be 6 trillion dollars until 2021, where security vulnerabilities in web applications increasing every year as new vulnerabilities 5000-7000 emerge [4].

Although there is a legal deterrent to cyber-attacks, preventing attacks, finding gaps and threats and addressing them before they occur is much less costly. Web applications assessment tools provide us with important facilities to identify any gaps or vulnerabilities helping developers avoid many important errors in the development phase. Mainly there are two approaches to assessment web applications [5], Black-Box which launch many attacks towards web application (we only need to know URL of that application) and Whit-Box approach that cares about analyzing the source code of the targeted application.

**Manuscript published on 30 March 2019.**
*Correspondence Author(s)
**Mohamad Ahmad Swead**, Web Sciences, Syrian Virtual University, Syria.
**Muhammad Mazen Almustafa**, Web Sciences, Syrian Virtual University, Syria.

There are already many web application assessment tools, commercial such as Acunetix which crawl and perform Black-Box hacking techniques to test for SQL Injection, XSS, XXE, SSRF, Host Header Injection and over 4500 other web vulnerabilities [6], Nessus is another tool with more features comparing to Acunetix such as auditing cloud infrastructure, basic network scan and many more [7]. As for open source tools, top 4 tools are [8]; WAPITI, Python-based that looking for scripts and forms in order to check injection threats, ZED attack proxy, developed by OWASP for beginners, VEGA mainly looking for SQL injection and XSS vulnerabilities, W3AF can be used to address more than 200 vulnerability such as blind SQL and buffer overflow.

## II. VULNERABILITIES BACKGROUND

Web applications vulnerabilities can be categorized into three categories [9] [10];

- Injection vulnerabilities such as SQL injection, Cross-site scripting XSS which happened mostly due to missing input validation.
- Session management vulnerabilities such as cross-site request forgery CSRF where attackers use this vulnerability to hijack session between client and server [8].
- Business logic vulnerabilities such as parameters tampering which can lead to injection attacks also.

### A. *SQL Injection*

The first public discussions of SQL injection started appearing around 1998 [11], SQL injection is one of the most important vulnerabilities in the world where the attackers have the ability to execute malicious SQL queries to control database of targeted application [4].

#### *1) SQL injection mechanism*

Most Web applications use databases that represent the application's black box. These applications rely on a three-tier structure: user, Web application, and database server (black box in this structure). The user sends an HTTP request to the web application. This request may contain an SQL query. In turn, the application communicates with the database server, which in turn responds to those requests and returns the results to the web application [12]. The attacker can create SQL queries for information, deletion or modification. Then the latter (web application) returns the results to the user as in the following figure:

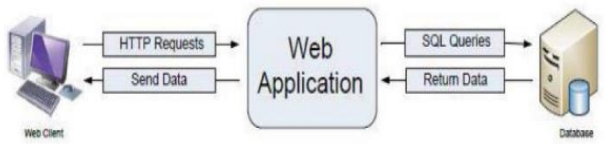


**Figure 2. Web application mechanism**

The following programming code is used for login authentication;

string uname = Username.Text;

string passwd = Pwd.Text;

sql = "SELECT id FROM users WHERE username='" + uname + "' AND password='" + passwd + "'"

database.execute(sql)

this code is vulnerable towards SQL injection by injecting malicious queries as below;

SELECT id FROM users WHERE username='username' AND password='password' OR 1=1'

So, the attacker will be able to bypass password and login to the application database without being authenticated [13].

#### *2) Common SQL injection types*

- Tautology attacks (Classic SQL): in this kind of attacks attacker aim to inject malicious queries with conditional statements to make it always true [13] [14]. The consequences of this type of attacks are used to bypass authentication.
- Union attacks: in this type of attacks, a malicious query is merged with a normal query using UNION in order to get information from other tables i.e.

SELETC * FROM users WHERE id='' UNION SELECT pwd from user_info where id='12' and pwd=''

- Error-based attacks [14]: attackers depending on error messages to get information about database structure to use it in another type of attacks.
- Out-of-band attacks [14]: is not very common, mostly because it depends on features being enabled on the database server being used by the web application. Out-of-band SQL Injection occurs when an attacker is unable to use the same channel to launch the attack and gather results.

#### *3) Preventing SQL injection*

In order to protect against SQL injection attacks, it is necessary to check database-oriented queries and accept only input-validated entries. To protect against blind SQL injection attacks, error messages must be disabled at the database level.

### B. *Cross-Site scripting XSS*

One of the most critical security vulnerabilities facing web applications usually happened due to web application failing to validate user input before returning results to the user. Ignoring user validation could allow the attacker to use malicious code which might be sent to other users and unexpectedly executed by their browsers, resulting in privacy breaches and other damages in the form of data leaks [15] [16] [1].

#### *1) Cross-Site scripting XSS mechanism*

To run malicious JavaScript scripts in the victim's browser, an attacker must find a gap or a way to inject a malicious load into a Web page visited by the victim. Here, the attacker can benefit from social engineering techniques to persuade the victim to visit a web page that contains vulnerable content and infected with malicious Java code [17]. In order to execute an XSS attack, Web pages that contain gaps must include the user's (victim) entries on their pages, in which an attacker can enter a set of characters that are used on the Web page and treated as codes in the victim's browser.

#### *2) Cross-Site scripting XSS types*

- Stored XSS one of the most damaging type of XSS it's also called persistent XSS [19], attackers inject a script (referred to as the payload) that is permanently stored on the target application. A classic example is a malicious script inserted by an attacker in a comment field on a blog or in a forum post [18].


Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

• Reflected XSS the attacker's payload script has to be part of the request which is sent to the web server and reflected back in such a way that the HTTP response includes the payload from the HTTP request [20]. Using Phishing emails and other social engineering techniques, the attacker lures the victim to inadvertently make a request to the server which contains the XSS payload and ends-up executing the script that gets reflected and executed inside the browser.

• DOM -based XSS is an advanced type of XSS attack which is made possible when the web application's client-side scripts write user-provided data to the DOM. The data is subsequently read from the DOM by the web application and outputted to the browser. If the data is incorrectly handled, an attacker can inject a payload, which will be stored as part of the DOM and executed when the data is read back from the DOM [16].

*3) Preventing Cross-Site scripting*

Same protection rules from SQL injection must be applied here also by validating user inputs. Proper output encryption will ensure that the browser deals with content that is potentially dangerous as text and not as active content that can be implemented. The best way to avoid DOM-based XSS is to make sure that All inputs from the user are passed to the server for validation. However, you cannot avoid using variables in the DOM, so the input validation should occur in the script itself.

## III. MOTIVATION

Recently, web application developers have resorted to encrypting all hidden and non-hidden transactions based on symmetric and asymmetric encryption algorithms, storing them in the database in order to provide a higher level of protection for those applications. However, vulnerabilities still exit by the ability to decrypt these values and launch injection attacks. In this research, we will focus on developing a methodology to assessment web applications by extracting hidden parameters, checking if it's encrypted or not and trying to decrypt it using the black-box approach. Research objectives can be summarized as the following;

• Crawling on web application URL in order to address vulnerabilities related to encrypted hidden parameters which lead to injection vulnerabilities such as SQL injection and XSS.

• Checking JavaScript in order to make sure that there are no malicious scripts which lead to XSS vulnerabilities.

## IV. PROPOSED METHODOLOGY

The objective of this research is developing a methodology to detect web applications vulnerabilities related to encrypted hidden parameters. Developers might tend to encrypt parameters using one of the following algorithms, MD5, SHA1, SHA2, AES.

A prototype called DEHP is implemented based on a proposed approach using the black-box technique. This section describes the architecture of the prototype.

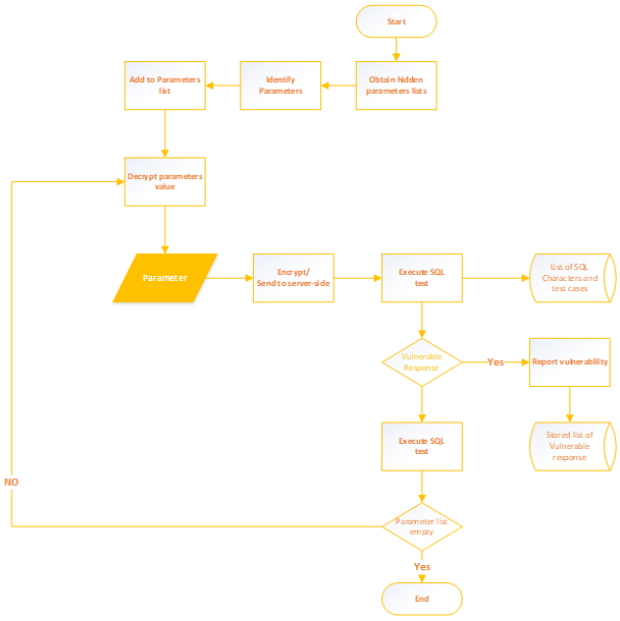


**Figure 3. DEHP Architecture**

### A. DEHP Modules

In this section, we will discuss in details DEHP main modules:

*1) DEHP Crawler*

The crawler is the main body of this prototype due to its role in visiting a number of web pages in targeted URL, extracting all included URLs and analyze it. DEHP is using traditional crawlers where it is starting from targeted URL, extracting all hyperlinks those URLs where the HTML resource type is specified by the request header and add it to the list of URLs to crawl and remove duplicate in the final stage of crawling.

Main functions of DEHP crawler are as follows;

- Process the next URL link from the list queue, connect and send a request to the Web server that is crawled via HTTP or HTTPS.
- Extract new links from the web page and add them to the list of links.
- Avoid requests for any sources containing "?" When requested to avoid a crawl trap that causes the crawler to enter an endless loop of loading and crawling.

DEHP is using regular expression algorithm where the HTML statements and text are parsed for each crawled page to extract the hyperlinks in the tag "<a>" and "<meta>" as below code

```
string[] urls;
List<string> ls = new List<string>();
var web = new HtmlWeb();
var doc = web.Load(u);
var temd = doc.DocumentNode.SelectNodes("//a/@href");
if(temd != null)
foreach (HtmlNode nodeitem in temd)
{
If(nodeitem.Attributes["href"].Value != null)
If(!ls.Contains(nodeitem.Attributes["href"].Value))
      ls.Add(nodeitem.Attributes["href"].Value);
}
```


Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

urls = ls.ToArray();

Below figure clarify DEHP crawler mechanism;

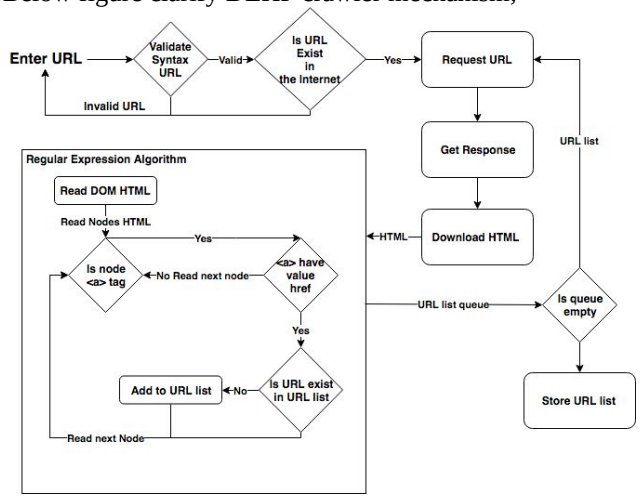


Figure 4. Traditional crawler in DEHP

*2) Encrypted hidden parameters module*

This module is used to detect injection vulnerabilities by checking hidden parameters value if it is encrypted using one of the hashing functions MD5, SHA1, SHA2 or not. <input> Tags are extracted from HTML page then the value of this parameter is being checked whether it's encrypted or not. If the value is encrypted DEHP launch one of two types of attacks in order to decrypt value;

- Brute-Force attack depending on English characters' array and special characters. The degree of complexity in this method is O(MN) in the worst case and O(M) in the best case. Thus, an application works on each hash function and the aes-256 decoding algorithm to find the K and IV key (Initial Vector) to solve the code and check the value if it's INT, Float, or text. The complexity of Brute-Force attack algorithm in DEHP is MA(F_md5(O(MN))) + MA(F_sha1(O(MN))) + MA(F_Sha256(O(MN))) + MA(Dec_aes(O(MN)))

- Dictionary attack we recommend this type of attacks due to its complexity which it less that Brute-Force attack where DEHP will match a large set of commonly used words which stored in the database in both forms, plain text and encrypted one (SHA1, SHA2, MD5). The complexity of this algorithm is MA(R) * Dec_aes() where R = Count of Records in the database. Below simple code clarifies work process of this module;

```
var tempdoc1 =hdurl.DocumentNode.SelectNodes("//input[@type=\"hidden\"]");
foreach (HtmlNode node in tempdoc1)
{
sss = node.Attributes["value"].Value;
dt = db_c.getquery(@"Select * from Ptable where sha1_hash_text like '" + sss + "' or md5_hash_text like '" + sss + "' or sha256 like '" + sss + "'");
dec_text = crc.DecryptStringFromBytes_Aes(Encoding.ASCII.GetBytes(sss), Encoding.ASCII.GetBytes(p_text), Encoding.ASCII.GetBytes(p_text));
}
check_the_result();
```

*3) Blind SQL injection module*

The attacker in SQL blind injecting values into a database to get true or false value or error message so it can be exploited. The injection of values into a database can be done by inserting incorrect text into SQL query or by adding a valid SQL query to an entry where no input validation is performed to be used to log in, enter, modify or even delete a value in a database, or even delete a table, or know tables and structure of the database to steal data.

DEHP extracts all (username) and (password) nodes from scanned HTML pages as below:

```
<input type="text" name="username" />
<input type="password" name="password" />
var nodecol1 = hd.DocumentNode.SelectNodes("//input[@name=\"username\"]");
var nodecol = hd.DocumentNode.SelectNodes("//input[@type=\"password\"]");
```

DEHP preparing web request using GET or POST to launch attack towards extracted nodes and analyzing response depending on regular expression algorithm as below pseudo code;

```
MyWebRequest myRequest = new MyWebRequest(uri, "POST", value);
string result = myRequest.GetResponse();
if (result.Contains("successfully")|| result.Contains("success")
|| result.Contains("weclome" + usernametamp)|| result.Contains("error")
|| !result.Contains("invalid") || !result.Contains("password"))
return true;
```

*4) XSS vulnerability module*

DEHP checks if there are any vulnerabilities related to dealing with the database using JavaScript where attacker can exploit this vulnerability to manipulate database values or know structure of the database which can be exploited to launch SQL injections attacks also. Checking is done by parsing HTML pages of requested URL and looking for any commands to connect to database,

querying or modifying database to limit the frequency of this kind of vulnerabilities as in the following pseudo code:

```
if (sss.ToLower().IndexOf("insert into") >= 0)
c_xss++;
if (sss.ToLower().IndexOf("createconnection") >= 0)
c_xss++;
if (sss.ToLower().IndexOf("insert into") >= 0)
c_xss++;
var t1 = hdurl.DocumentNode.SelectNodes("//script");
if (t1 != null)
foreach (var node1 in t1)
{
string s1 = node1.InnerHtml;
if (s1.ToLower().IndexOf("createconnection") >= 0)
  c_xss++;
if (s1.ToLower().IndexOf("insert into") >= 0)
  c_xss++;
}
```


Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

*5) HTTPS certificate module*

DEHP checks the digital certificate when using HTTPS protocol to connect to the server where the connection is encrypted. DEHP check authenticity of the digital certificate according to the X.509 standard.

```
if (sslPolicyErrors == SslPolicyErrors.None)
{
return true;
}
else
{
return false;
}
```

DEHP database

DEHP database consists of one main table storing key words in plain text format and MD5, SHA1, SHA2 format to ease search operation for guessing encrypted word and try to decrypt it.

## B. DEHP Mechanism

The following flow chart describes DEHP mechanism:

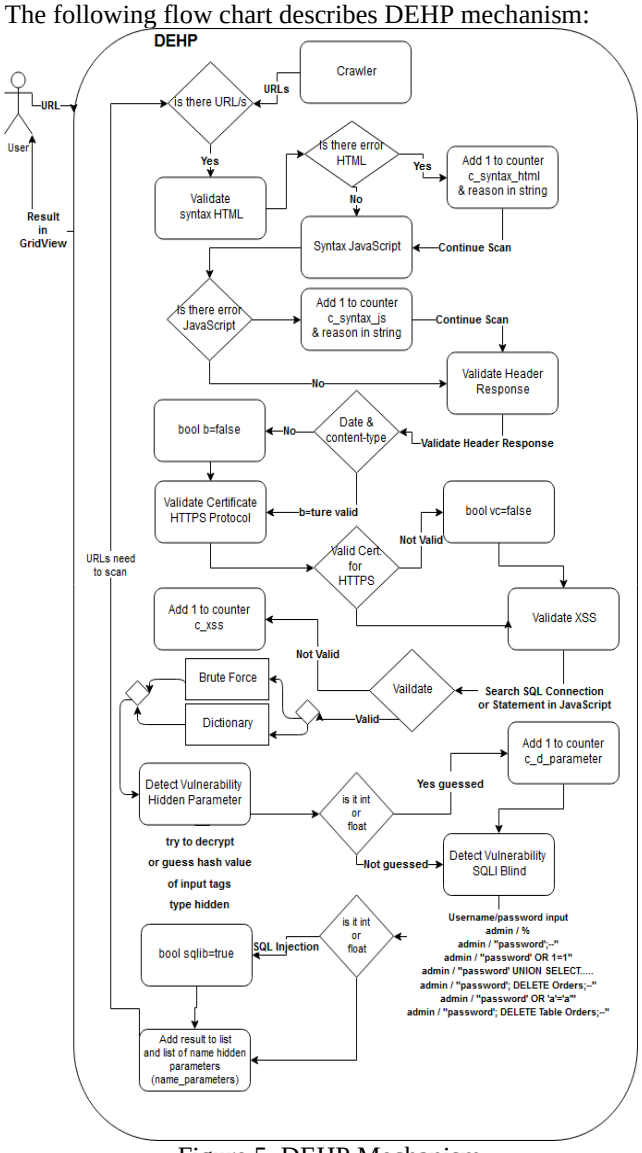


Figure 5. DEHP Mechanism

- Importing Excel file to database

DEHP importing excel file which contains a set of the most common words then encrypt these words using hashing functions MD5, SHA1, SHA2. The following figure shows interface of importing excel file to DEHP database.

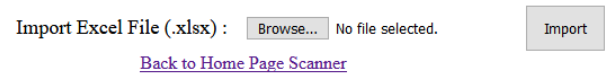


**Figure 6: Import excel file**

- Scanning web application

In order to scan web application, the user needs to enter URL of that application and choose one of two kinds of attacks (Brute-Force attack, Dictionary attack) as the following figure;

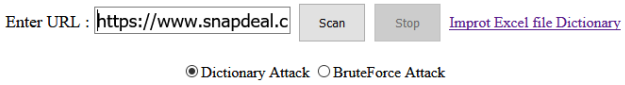


**Figure 7. Scan web application.**

## V. RESULTS

### A. *Figures and Tables*

Results of DEHP have been compared with previously mentioned open source software Wapiti, Skipfish, Zed Attack Proxy, and W3af. The results are explained in below figure;

| Scanner | Version | WebScanTest | TestPHP | TestASPnet | Rank(speed) | Blind SQL | SQLI cases | XSS cases | Interface | features | live feedback | Detailed | user | configuration |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Wapiti | 2.3.0 | 6:20:00 | 1:30:00 | 3:30:00 | third | No | 131 | 45 | GUI | 10 modules for 9 vulnerability type | yes | yes | Easy | Basic |
| Skipfish | 2.1 | 0:11:06 | 0:27:40 | 0:11:30 | second | No | 104 | 63 | GUI | Different scanning modes and dictionaries | no | yes | Easy | Basic |
| Zed Attack Proxy | 2.7.0 | none | none | none | none | Fuzzing | 13 | 26 | GUI | 15 modules: Web sockets, ajax spiders, man-in-the-middle etc | no | yes | Easy | Basic |
| W3af | 1 | fast scan 0:10:25 | fast scan 0:03:12 | fast scan 0:12:33 | second | Yes | 94 | 12 | GUI / web base | Common vulnerabilities such as XSS, SQLI and uncommon | yes | yes | Easy | Flexible |
| DEHP | 1 | 0:03:06 | 0:02:07 | 0:02:25 | first | yes | (sha1, sha256, aes, md5) | 1 | GUI / web base | 5 modules (syntax, header, cer., SQLI encrypted hidden parameter, XSS) | no | yes | Easy | Flexible |

Figure 8. Comparison results

Depending on the previous table, we can conclude results related to scanning time as below;

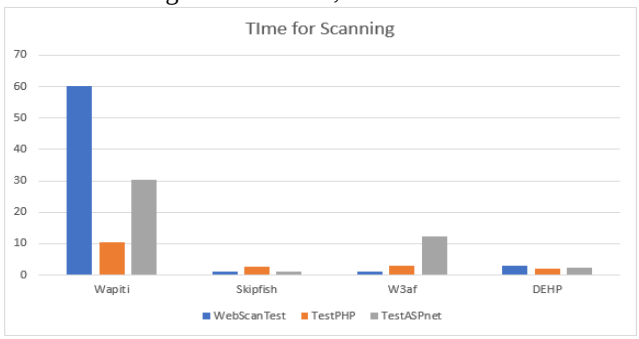


**Figure 9. Scanning time**


Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

## VI. DISCUSSION

From the previous results and statistics for the number and types of vulnerabilities and time, we find that they are different ratios due to the uniqueness of each application in a way to crawl and scan for the various vulnerabilities while DEHP in this research characterized by simple way crawling and scan for HTML errors, XSS, SQLI blind and finally, detecting the hidden encrypted parameter that may lead to an SQLI vulnerability. One of DEHP strength is the looking for any encrypted hidden parameters and trying to decrypt it. The database needs to be extended, and we have faced many difficulties in testing Brute-Force attack due to lack of required resources.

## VII. CONCLUSION

Recently, validation of user entries has become a real standard in web application programming. Ignoring this step makes web application loses an important part of its security (Doup´e, 2014). As user-side scripting such as JavaScript, for example, emerged and resulted in a fast transition to validate user input in the browser itself before sending data to the server, which contributed to reducing the burden on the server, both in terms of time or processing. Security vulnerabilities in web applications can be categorized generally to two main categories, injection-related security vulnerabilities and application logic vulnerabilities which also can lead to injection vulnerabilities. Previous researches and studies related to application logic and injection didn't care about encrypted hidden parameters, which we focus on in our research using the Black-Box approach. During our research, we had some difficulties related to performance and laws even where we were limited in testing in order to respect laws. As for future work, DEHP can be developed by adding some extra features such as looking for cookies vulnerabilities when using SSL or TLS connection. Enhance DEHP performance by enhancing dictionary attack algorithm finding suitable test bed environment for Brute-Force attack module.

## ACKNOWLEDGMENT

We want to thank everyone supports us during our work, especially our families.

## REFERENCES


1. J. Fonseca, M. Vieira and H. Madeira, "Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks," in Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on, 2007.
2. J. Fonseca, N. Seixas, M. Vieira and H. Madeira, "Analysis of field data on web security vulnerabilities," IEEE transactions on dependable and secure computing, vol. 11, no. 2, pp. 89-100, 2014.
3. W. Security, "WEB APPLICATIONS Security Statistics Report," WhiteHat Security, 2016.
4. H. One, "The Haker-Powered security report," Hacker One, 2018.
5. C. Zhu, Experimental study of vulnerabilities in a web application, Aalto University, 2017.
6. ACUNTIX. [Online]. Available: https://www.acunetix.com/. [Accessed 1 10 2018].
7. Nessus. [Online]. Available: https://www.tenable.com/products/nessus/nessus-professional. [Accessed 1 10 2018].
8. OWASP, 1 October 2018. [Online]. Available: https://www.owasp.org/index.php/Main_Page.
9. P. S. T. A. P. A. R. P. G. Deepa, "DetLogic: A black-box approach for detecting logic vulnerabilities in web applications," Journal of Network and Computer Applications, 2017.
10. X. Li and Y. Xue, "A survey on server-side approaches to securing web applications," ACM Computing Surveys (CSUR), vol. 46, no. 4, p. 54, 2014.
11. A. a. V. Ciampa, C. A. a. D. Penta and Massimiliano, "A heuristic-based approach for detecting SQL-injection vulnerabilities in Web applications," in Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, 2010.
12. A. L. Doup´e, Advanced Automated Web Application, Califronia, 2014.
13. N. Khoury, P. Zavarsky, D. Lindskog and R. Ruhl, "An analysis of black-box web application security scanners against stored SQL injection," in Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on, 2011.
14. C. Sharma and S. Jain, "Analysis and classification of SQL injection vulnerabilities and attacks on web applications," in Advances in Engineering and Technology Research (ICAETR), 2014 International Conference on, 2014.
15. M. E. Ruse and S. Basu, "Detecting cross-site scripting vulnerability using concolic testing," in Information Technology: New Generations (ITNG), 2013 Tenth International Conference on, 2013.
16. C. Baojiang, L. Baolian and H. Tingting, "Reverse analysis method of static XSS defect detection technique based on database query language," in P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on, 2014.
17. L. B. H. T. Cui Baojiang, "Reverse Analysis Method of Static XSS Defect Detection Technique Based on Database Query Language," in 2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 2014.
18. M. Parvez, P. Zavarsky and N. Khoury, "Analysis of effectiveness of black-box web application scanners in detection of stored SQL injection and stored XSS vulnerabilities," in Internet Technology and Secured Transactions (ICITST), 2015 10th International Conference for, 2015.
19. X. Guo, S. Jin and Y. Zhang, "XSS vulnerability detection using optimized attack vector repertory," in Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2015 International Conference on, 2015.
20. S. B. J. G. V. J. Bhawna Mewara, "Enhanced Browser Defense for Reflected Cross-Site Scripting," IEEE, 2014.
21. M. F. Y. Zainab S. Alwan, "Detection and Prevention of SQL Injection Attack: A Survey," International Journal of Computer Science and Mobile Computing, vol. 6, no. 8, pp. 5-17, 2017.
22. Y. K. K. K. Yusuke Takamatsu, "Automated Detection of Session Fixation Vulnerabilities," in Automated Detection of Session Fixation Vulnerabilities, 2010, pp. 1191-1192.
23. A. Yeole and B. Meshram, "Analysis of different technique for detection of SQL injection," in Proceedings of the International Conference & Workshop on Emerging Trends in Technology, 2011.
24. S.-K. H. T.-P. L. C.-H. T. Yao-Wen Huang, "Web Application Security Assessment by Fault Injection," in WWW 2003, Budapest, Hungary, 2003.
25. Y. X. Xiaowei Li, "A survey on server-side approaches to securing web applications," ACM, p. 29, 2014.
26. J. V. A. O. William G.J. Halfond, "A Classification of SQL Injection Attacks and Countermeasures," IEEE, 2005.
27. Wikipedia, "Microsoft SQL Server," [Online]. Available: https://en.wikipedia.org/wiki/Microsoft_SQL_Server. [Accessed 30 12 2018].
28. Wikipedia, "C Sharp," Microsoft, [Online]. Available: https://en.wikipedia.org/wiki/C_Sharp_(programming_language). [Accessed 30 12 2018].
29. Wikipedia, "ASP.NET," Microsoft, [Online]. Available: https://en.wikipedia.org/wiki/ASP.NET. [Accessed 30 12 2018].
30. wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/HTTPS. [Accessed 10 4 2018].
31. webscantest. [Online]. Available: http://webscantest.com/report/Vulnerabilities_1.html. [Accessed 7 1 2019].




*Published By:*
*Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP)*

32. X. Wang, L. Wang, G. Wei, D. Zhang and Y. Yang, "Hidden web crawling for SQL injection detection," in Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on, 2010.
33. R. D. R. C. D. Trupti V.U dapure, "Study of Web Crawler and its Different Types," IOSR Journal of Computer Engineering (IOSR, vol. 16, no. 1, pp. 1-5, 2014.
34. Symantec, "Internet Security Threats Report," Symantec, 2017.
35. Sucuri, "Hacked Website Report," Sucuri, 2017.
36. M. V. Studio, "Wikipedia," Microsoft, [Online]. Available: https://en.wikipedia.org/wiki/Microsoft_Visual_Studio. [Accessed 30 12 2018].
37. E. K. C. K. a. N. J. Stefan Kals, "SecuBat: A Web Vulnerability Scanner," International World Wide Web Conference Committee, pp. 247-256, 2006.



Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)