EXPLORING SQL INJECTION VULNERABILITIES USING ARTIFICIAL BEE COLONY

Kevin Baptista, Anabela Bernardino and Eugénia Bernardino
Computer Science and Communication Research Center (CIIC)
School of Technology and Management, Polytechnic of Leiria, Portugal

ABSTRACT

Over the last couple of decades, there has been an enormous growth in technologies and services available on the internet. This growth must take security into account, although due to the increase in complexity of systems this is not an easy task. Nowadays, hardly any organization may say with certainty that their system is secure. The Open Web Application Security listed "Injection" as the most security risk for web applications in 2020. There are many automated tools to assist professionals in the field, in order to identify this vulnerability. However, keeping these tools up to date has proven to be a challenge. Therefore, there has been some interest in applying Artificial Intelligence (AI) in this field. In this paper, we propose an approach to detect SQL injection vulnerabilities in the source code, using Artificial Bee Colony (ABC). To test this approach empirically we used web applications purposefully vulnerable as Bricks, bWAPP, and Twitterlike. Simulation results verify the effectiveness of the ABC algorithm.

KEYWORDS

Artificial Bee Colony, SQL Injection, Vulnerabilities in Web Applications

1. INTRODUCTION

The growth and usage of the Internet worldwide (Barros and Dumas, 2006), has lead to attacks on information systems that are becoming every year more sophisticated and catastrophic, resulting many times in loss of personal data and, in extreme cases, human lives.

The Open Web Application Security (OWASP) listed the top 10 web application security risks for 2020 as injection, broken authentication, sensitive data exposure, XML external entities, broken access control, security misconfiguration, cross-site scripting, insecure deserialization, using components with known vulnerabilities and insufficient logging and monitoring (Stock et al., n.d.).

The focus in this paper is on injection, more precisely, SQL Injection. OWASP defines SQL Injection as a vulnerability that occurs when untrusted data is sent to the system as part of a query. The main goal for the attacker is to trick the interpreter into executing unintended queries in order to execute unauthorized actions like obtained unauthorized data.

Artificial Intelligence (AI) techniques have been successfully applied in many areas of software engineering (Columbus, 2008). In order to detect vulnerabilities of SQL Injection, we developed an automated tool, based on the application of an Artificial Bee Colony (ABC) algorithm. ABC is a simple and effective global optimization algorithm that has been successfully applied to several optimization problems of various fields (Karaboga et al., 2014). The main purpose of this tool is to be used in a white box scenario, having access to the code base. Thus, it could help developers to find potential vulnerabilities in their codebase.

To empirically evaluate our presented approach, we used several open-source projects that are known to have certain vulnerabilities, such as Bricks, bWAPP, and Twitterlike.

The rest of the paper is organized as follows. Section 2 presents some related work. Section 3 describes the identified problem. In Section 4, it is described how the problem was modelled in order to be used as an optimization problem. Section 5 describes the proposed ABC algorithm. Section 6 discussed the various experiments conducted to empirically validate the approach taken as well as the specifications used to conduct the analysis and the results obtained. Section 7 concludes this paper.

2. RELATED WORK

Mckinnel et al. (2019), make a comparative study of several artificial intelligence algorithms in exploiting vulnerabilities. In their study, the authors compiled several works in the area, in order to compare several algorithms, including unsupervised algorithms, Reinforcement Learning, Genetic Algorithms (GAs), among others. The authors conclude that the GAs performs better over time due to the evolutionary nature of generations. They suggest that its applicability needs to take into account a good definition of the fitness function in order to obtain better results. They recommend solutions designed for specific systems, instead of generic solutions since these more generic solutions only exploit shallower and simpler vulnerabilities.

Manual penetration tests, although effective, can hardly meet all security requirements that are constantly changing and evolving (Niculae, 2018). Furthermore, they require specialized knowledge which, in addition to presenting a high cost, is typically slower. The alternative is automated tools that, although faster, often do not adapt to the context and uniqueness of each application. In (Niculae, 2018), the author developed a reinforcement learning strategy capable of compromising a system faster than a brute force and random approach. This concluded that it is possible to build an agent capable of learning and evolving over time so that it can penetrate a network. Its effectiveness was equal to that of human capacity. Finally, he concludes that although the initial objective was long, there are still several directions to be explored. From the use of different algorithms for both an offensive (red team) and a defensive (blue team) security perspective. It suggests the application of game theory concepts (Nguyen et al., 2016), especially treating a problem like a Stackelber Security Game. These techniques have been successfully applied in various security domains such as finding the optimal allocation for airport security given the attackers' knowledge.

Alenezi and Javed (2016) analyzed several open-source projects in order to identify vulnerabilities. They concluded that most of these errors are due to negligence on the part of the people who developed the applications as well as the use of bad practices. To solve this problem, the authors suggest the development of a framework that encourages programmers to follow good practices and detect possible flaws in the code.

In Tripathi et al. (2018), the authors propose a solution to detect XSS (cross-site scripting) vulnerabilities based on the use of GAs as well as a proposal to remove the vulnerabilities found during the detection phase. The aim was to find as many vulnerabilities as possible with as few tests as possible. The results obtained were compared with other static analysis tools. As future work, they suggest the application of this technique in applications from other areas as well as to three types of XSS: persistent, reflected and based on DOM.

3. PROBLEM DEFINITION

Security vulnerabilities are flaws in web applications that allow attackers to execute actions that should not be allowed. SQL Injection is a vulnerability that happens when the attacker is able to execute SQL commands that should not be possible by sending very specific values to the server. The primary reason for this vulnerability to happen is the lack of input validation employed in the applications.

Most web applications have some sort of forms to be filled by the user, either to authenticate, to register, or to search. In consequence, the server receives the input sent by the client and executes some business rules to it. Commonly, the server communicates with a database (SQL or NoSQL) to save the data. During this process, the developer must have some attention to save the data in a safe manner, otherwise, it can be exploited by an attacker.

Let us consider the authentication in a web application. In this scenario, the user would be presented with a form with two input fields: one for the username and the other for the password. For the sake of simplicity, let us ignore other authentication methods available nowadays. On the server, it would be expected to see a query similar as follows:

```
SELECT * FROM users
WHERE username = '$username' AND password=md5('$password');
```

MD5 is a cryptographic algorithm, often used to store passwords in a database. Even though, this algorithm is no longer safe, due to be easily cracked by brute force and having dictionary tables for it (Zheng and Jin, 2012). Nevertheless, this is not an issue for the current problem. Any algorithm with salt could be used to represent this problem, however, this one was used due to its simplicity.

The users fill the form with username “John” and password “example”, then we would have the following execution:

```
SELECT * FROM users
WHERE username = 'John' AND password=md5('example');
```

This query searches for a user that has the username "John" and the hash of the password "example". If the result of the search is not empty, then it would authenticate the user.

The problem comes when an attacker sends a value that would trick the system. Consider that the attacker fills username as "any" and password as "x') OR 1=1; --". Then the query would look like the following:

```
SELECT * FROM users
WHERE username = 'any' AND password=md5('x') OR 1=1; --';
```

Simplified we would find three conditions: *username='any'* which is false; *password=md5('x')* which is also false; but then *1=1* is true. In other words, we would have:

```
false AND false OR true
false OR true
true
```

So, it does not matter what the client sends as username or password, if they were able to manipulate the query to execute a third condition with "OR true". The consequence would be that a user can be able to login into the system without valid credentials. In general, a successful SQL injection attack attempts different techniques, such as the one demonstrated above in order to carry out a successful attack.

SQL injection could have catastrophic consequences and is the number one vulnerability risk according to OWASP (Stock et al., n.d.). However, there are a couple of measures that can be implemented in order to prevent this injection. The preferred option is to use a safe API, which avoids the use of an interpreter entirely or provides a parameterized interface, or even migrates to an Object Relational Mapping tool (ORM).

4. PROBLEM REPRESENTATION

In order to use an ABC as an optimization algorithm to find SQL Injection, the process was divided into two steps: (1) identification of all SQL queries (Figure 1); (2) use of ABC to generate attack vectors to be injected in the queries (Figure 2).

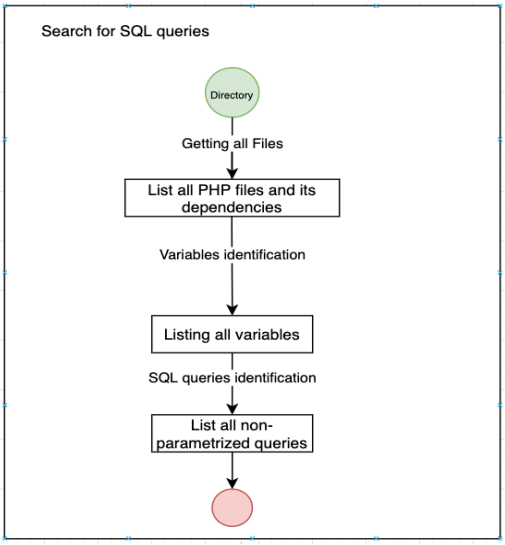

Figure 1. First step in the process: Searching for Queries

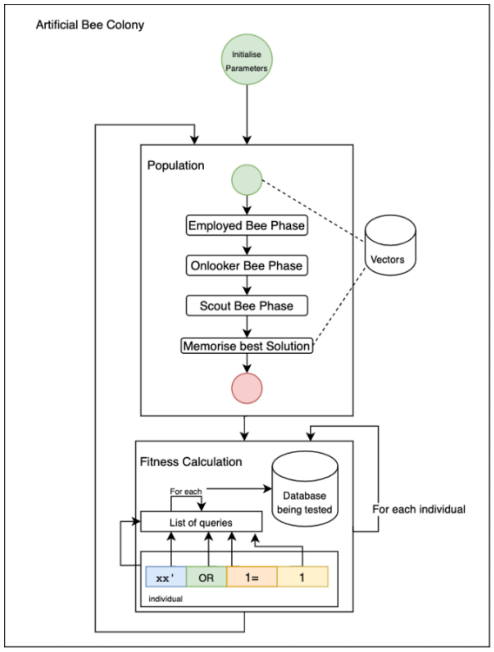

Figure 2. Second step in the process: Finding Best Solution

4.1 Search for SQL Queries & Find Best Solution

In Figure 1, we can see an overview of the process of searching for SQL queries, where the main goal is to obtain all non-parametrized queries. In order to obtain this goal, first, it is necessary to perform a search to list all PHP files recursively in a given folder. Afterwards, all variables in the code are indexed and their history is kept. This step is crucial to capture SQL queries that are parametrized, but still vulnerable because the vulnerabilities occurred before in the code. These queries and all non-parametrized queries are kept in a list to be used in the second phase by the Artificial Bee Colony algorithm.

The next step occurs in the ABC domain as we can observe in Figure 2. The main goal of this step is to find a vector that could compromise one of the queries listed in the previous step. So, the algorithm starts by initializing all the needed parameters, creates a population, and executes all steps in the ABC.

In order to inject vectors in the queries, which are selected by the algorithm, a dataset is provided beforehand and stored in a database. This dataset was built based on Friedl (2017) and Mishra (n.d.).

4.2 Individual Representation

In our representation, individuals are derived from SQL injection database which was constructed based on different resources (Friedl, 2017; Mishra, n.d.). Each individual is made up of a set of four genes. Each gene is a String. In Figure 3, there is a possible representation for the individual. Each gene could be either a logic operator (for example OR, AND, NOT, etc.) or a value (for example: "xx'", "1", "1=", " -- "). The key point of encoding an individual as a list of strings is that, when we calculate the fitness of an individual, we will use this string as multiple attack vectors.

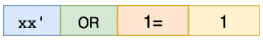

Figure 3. Example of an individual

4.3 Fitness

As represented in Figure 2, each gene in the individual is going to be tested as an attack vector, and also all of them as one attack. To illustrate this idea, let us consider again the individual in Figure 3 and the code presented in Figure 4.

```
<?php
        $uagent = $_SERVER['HTTP_USER_AGENT'];
        $sql= "SELECT * FROM users WHERE ua='$uagent' ";
?>
```

Figure 4. Example of PHP code

The queries executed for this scenario are illustrated in Table 1 and as we can observe, an individual which has 4 genes executes 5 queries. For this example, only one of them is valid and vulnerable.

Table 1. Executed queries

Query	Valid / Invalid / vulnerable	Success
`SELECT * FROM users WHERE ua='xx''`	Invalid Query. Not vulnerable.	No
`SELECT * FROM users WHERE ua='OR'`	Valid Query. Not vulnerable.	No
`SELECT * FROM users WHERE ua='1='`	Valid Query. Not vulnerable.	No
`SELECT * FROM users WHERE ua='1'`	Valid Query. Not vulnerable.	No
`SELECT * FROM users WHERE ua='xx' OR 1=1`	Valid Query. Vulnerable.	Yes

The fitness function used for this problem is as follows (Eq. 1):

$$fitness(i) = \frac{U_{vul} * 5 + G_{vul} * 2}{totalGenes} \quad (1)$$

where i is the individual being tested, $totalGenes$ is the total number of genes in an individual, U_{vul} is the number of unique vulnerabilities detected by the individual. G_{vul} is the number of genes that detected at least one vulnerability.

Fitness is used to evaluate the performance of an individual for a given problem. An individual with bigger fitness means that has better performance, or in other words, was able to crack successfully more queries.

Applying this fitness function to the example described above means we would get the fitness:

$$fitness = \frac{1 * 5 + 1 * 2}{4}$$

We have only analysed one query, thus at best the U_{vul} is one, which is the case. As can be seen, every gene tested individually did not get any success. On the other hand, when tested grouped it got one success, thus $G_{vul} = 1$. It is important to remember, that this example is very simple when we are testing only one SQL query. In empirical tests, there are several queries, so the success might be different.

5. ARTIFICIAL BEE COLONY

ABC was proposed by Karaboga (2005) to optimise numerical problems. ABC simulates the intelligent behaviour of a bee colony (Karaboga and Basturk, 2007a,b; Karaboga and Akay, 2009a,b). The minimal model of swarm-intelligent forage selection in a honey bee colony, that ABC algorithm adopts, is based on three kinds of bees: employed bees, onlooker bees, and scout bees (Karaboga and Akay, 2009a,b). To our knowledge, this work presents the first application of ABC to solve the problem presented in this paper.

5.1 ABC Steps

In order to apply ABC to a problem, some steps should be performed. Each iteration of the search consists of four steps (Karaboga and Akay, 2009a,b):

1) Sending the employed bees onto their food sources and evaluating their nectar amounts;
2) After sharing the nectar information of food sources, selecting food source regions by the onlookers and evaluating the nectar amount of these food sources;
3) Determining the scout bees and then sending them randomly onto possible new food sources;
4) Memorising the best food source.

These four steps are repeated through a predetermined number of iterations defined by the user. In a robust search process, the exploration and exploitation processes must be carried out together. In the ABC algorithm, while onlookers and employed bees carry out the exploitation process in the search space, the scouts control the exploration process. The main steps of the ABC algorithm are described in the next sections.

5.1.1 Initialisation of Parameters

The following parameters must be defined by the user: number of employed bees (*ne*), number of onlooker bees (*no>=ne*), number of modifications (*nm*), and maximum number of iterations (*mi*).

5.1.2 Food Source Position Initialisation

At the first step, ABC generates randomly an initial bee population.

5.1.3 Employed Bees Phase

Employed bees are responsible for: (1) exploiting the nectar sources explored before and (2) giving information to the other waiting bees (onlooker bees) in the hive about the quality of the food source site which they are exploiting. In this phase, each employed bee produces a new food source in its food source site and exploits the best source. In our implementation, the position of a food source represents a possible solution to the problem and the nectar amount of a food source corresponds to the associated solution quality (fitness – Eq. 1). The number of employed bees is exactly the number of initial solutions. The employed bees share the information related to the nectar of the food sources and their position with the onlooker bees. An artificial onlooker bee chooses a food source, depending on the probability value associated with that food source, pr_i. The probabilities are calculated using Eq 2.

$$totalFitness = \sum_{n=1}^{ne} fit_n \qquad pr_i = \frac{fit_i}{totalFitness} \tag{2}$$

5.1.4 Onlooker Bees Phase

Onlooker bees wait in the hive and decide which food source to exploit according to the information shared by the employed bees. In this phase, each onlooker bee selects a source depending on the quality (fitness value) of its solution, produces a new food source in the selected food source site and exploits the best source (the source with the best fitness value).

An onlooker bee evaluates the nectar information taken from all employed bees and chooses a food source with a probability related to its nectar amount. Our algorithm computes the number of onlooker bees, which will be sent to food sources of employed bees (Eq. 3), according to the previously determined probabilities:

$$no_i = pr_i * no \qquad no_i = \text{number of onlooker bees sent to food source } i. \tag{3}$$

A neighbour is obtained by performing multiple attempts to improve the solution, which length is specified as *nm* (number of modifications). The algorithm performs *nm* modifications to find a new position for the onlooker bee. A modification consists of changing the value of a position of the individual to another random value (a string, as represented in Figure 3). The algorithm repeats this process until at least one exchange with improvement is made or until the *nm* is reached. If the nectar amount of the solution is higher than the nectar amount of the previous one, the bee memorises the new position and forgets the old one.

5.1.5 Scout Bees Phase

Scouts randomly search the environment in order to find a new food source depending on an internal motivation, possible external clues or randomly. In this phase, the food source of which the nectar is abandoned by the bees is replaced with a new food source by the scouts. In our implementation, this is simulated by producing new solutions and replacing the worst employed bees. This means that the food sources with lower nectar amounts are abandoned.

The worst employed bees as many, as the number of scout bees in the population, are respectively compared with the scout solutions. If a scout bee is better than an employed bee, the employed bee is replaced with the scout bee. Otherwise, the employed bee is transferred to the next cycle without changes. In our implementation, we consider the number of scout bees equal to 10 % of the number of employed bees (see Eq. 4).

$$ns = 0.1 * ne \tag{4}$$

5.1.6 Best Solution Memorization

In this step the algorithm memorises the best solution achieved so far.

6. EXPERIMENTAL RESULTS

All experiments were performed on a Raspberry PI 4 Model B, 8GB of RAM, quad-core 64-bits of 1.5Ghz.

The results produced by ABC are compared with manual analysis and with another tool, Web Application Protection (WAP) - http://awap.sourceforge.net/- that uses a static analysis approach to detect vulnerabilities in web applications written in PHP.

As mentioned previously, we have first identified SQL Injection vulnerabilities manually. We consider our manual analysis as the true value. Table 2 shows the results from this manual analysis.

In order to obtain the best combination of parameters, several smoke tests were performed. Figure 5 shows the effect that the number of modifications has on the quality of a solution. On the y-axis we have the total number of vulnerabilities detected. In the Bricks and Twitterlike projects, there are no effect with the number of modifications studied. On the other hand, there is a slight improvement with 2 modifications on the bWAPP project. This is probably due to the overall size of the project. Both Bricks and Twitterlike are relatively small when compared with bWAPP.

The relation between population size and execution time could not be measured and compared accurately. One improvement done during the testing phase was to cache queries that were already known as vulnerable or as not vulnerable. This had a huge impact in the performance point of view of the research. Taking as an example a population of 100 individuals, for 100 generations, for 30 different seeds, in the bricks projects (which has 11 queries identified) and each individual, 4 genes, that would translate in 100*100*30*11*(4+1) = 16 500 000 queries done in the database for one set of parameters.

Table 2. Results from the manual analysis

Web Application	Total SQL injection
Bricks	12
bWAPP	56
Twitterlike	17

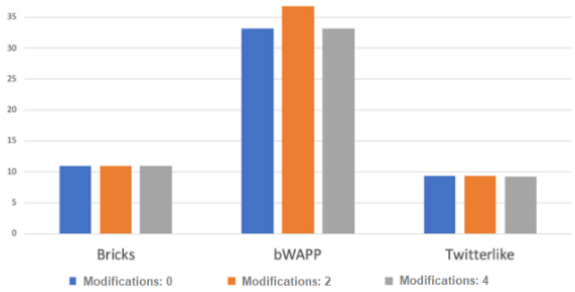

Figure 5. Impact of number of number of modifications on number vulnerabilities detected

The execution time that a query takes in a local database, depends on many factors, from the specifications of the machine to the complexity of the query. After some empirical tests, we conclude that our machine was taking around 130ms to return the result, which means that for the use case described before it would take almost 25 days to execute. Using a caching strategy, which stores the success or unsuccess of a query in memory, and using the same machine, a million access to one element of a map is taking around 80ms, which means that in average each access is taking $8 * e^{-5} ms$.

Taking into account the number of queries mentioned before, we reduce the time to 1320ms which is ~1.3seconds. As shown, this improvement was extremely efficient. Although now, parameters of the artificial bee colony have basically no impact on the execution time. When a query hits the cache it is extremely faster, which means the first sets of parameters to be explored will eventually be slower, because the cache is empty or almost empty. So, depending on the order we execute tests we can have situations where a bigger population is faster than a smaller one, due to the number of hits in the cache.

Table 3 illustrates the ABC parameters settings used that obtained the best values in all projects tested. Table 4 shows the results in terms of the vulnerabilities found with the ABC. Table 5 presents some examples of attack vectors found by this approach, that could be used in order to take advantage of SQL injection and corresponding examples of queries used with success. Table 6 shows the results when using WAP.

Table 3. Parameters that obtained best results

Parameter	Value
Max iterations	50
Number Employed bees	20
Number Onlooker bees	50
Number of modifications	2

Table 4. Results from Artificial Bee Colony

Web Application	Total SQL Injection Found
Bricks	11
bWAPP	47
Twitterlike	10

Table 5. Example of successfully vectors and query examples

Vector	Query executed
x‘=’x’ --	`SELECT * FROM users WHERE email = 'x'='x' --';`
x’=’x	`SELECT * FROM heroes WHERE login = 'x'='x';`

Table 6. Results using WAP

Web Application	Total SQL injection
Bricks	11
bWAPP	15
Twitterlike	5

As can be seen, with our approach most vulnerabilities are detected. For the case of bWAPP, only 50% of the vulnerabilities were detected with these parameters. This is probably due to the fact that an individual has a fixed genome in terms of size, which sometimes leads to invalid queries. An approach with a dynamic genome size could potentially bring better results in terms of total SQL injection vulnerabilities found.

When comparing with WAP tool, the ABC was able to identify more vulnerabilities. Only for the bricks use case, we got the same results when comparing to the same tool.

7. CONCLUSION

SQL injection is a current problem in web applications and can have serious implications and consequences. In order to detect those vulnerabilities, we presented an approach to detect these vulnerabilities in the code base, using a white-box approach. The problem was addressed using Artificial Bee Colony as a way to

optimize the search of potential vulnerabilities in the code. Dividing the process into two steps, starting with a searching phase for queries, then followed by an optimization search to find the best vectors, it was possible to obtain good results. As we could observe in Section 6, there was a slight improvement in results using the ABC when compared with static analysis.

The tool has been developed in Java, with support to analyse PHP applications, since PHP is one of the most used languages for web applications (Mishra, 2014), although, as other languages are gaining more presence online, the tool should be expanded to support multiple languages.

Another point that has a direct impact on the results obtained is the initial dataset given to the ABC. Having it in mind, it is crucial to expand this dataset in order to obtain better results and as mentioned previously, testing an individual with a dynamic genome could also bring some interesting results to the problem.

The scope of this article was constrained to SQL injection, but there are other injection problems and even other vulnerabilities that could potentially benefit from this approach.

ACKNOWLEDGEMENTS

This work was supported by national funds through the Portuguese Foundation for Science and Technology (FCT) under the project UIDB/04524/2020.

REFERENCES

Alenezi, M. and Javed, Y., 2016. Open source web application security: A static analysis approach. *Proceedings of the 2016 International Conference on Engineering and MIS*.

Barros, A.P. and Dumas, M., 2006. The Rise of Web Service Ecosystems. *In IT Professional*, 8 (5), pp. 31-37.

Columbus, L., 2018, January 12. 10 charts that will change your perspective on artificial intelligence's growth. *Forbes*.

Friedl, S., 2017, March 6. *SQL Injection Attacks by Example*. Retrieved Feb. 28, 2021, from http://www.unixwiz.net/techtips/sql-injection.html.

Karaboga, D. 2005. *An idea based on honey bee swarm for numerical optimization*, Technical report TR06. Erciyes University, Engineering Faculty, Computer Engineering Department.

Karaboga, D. and Akay, B., 2009a. Artificial Bee Colony (ABC), Harmony Search and Bees Algorithms on Numerical Optimization. *IPROMS 2009 Innovative Production Machines and Systems Virtual Conference*, Cardiff, UK.

Karaboga, D. and Akay, B., 2009b. A comparative study of Artificial Bee Colony algorithm. *In Applied Mathematics and Computation*, 214:108–32.

Karaboga, D. and Basturk, B., 2007a. Artificial Bee Colony (ABC) Optimization Algorithm for Solving Constrained Optimization Problems. *Proceedings of the 12th International Fuzzy Systems Association World Congress on Foundations of Fuzzy Logic and Soft Computing*. Berlin Heidelberg, Springer-Verlag, pp. 789–798.

Karaboga, D. and Basturk, B., 2007b. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm. *In Journal of Global Optimization*, 39(3), pp. 459–71.

Karaboga, D. et al, 2014. A comprehensive survey: artificial bee colony (ABC) algorithm and applications. *In Artificial Intelligence Review*, 42, pp. 21–57.

McKinnel, D.R. et al, 2019. A systematic literature review and meta-analysis on artificial intelligence in penetration testing and vulnerability assessment. *In: Computers & Electrical Engineering*, 75, pp. 175-188.

Mishra, A., 2014. Critical Comparison of PHP And ASP.NET For Web Development - ASP.NET & PHP. *In International Journal of scientific & Technology Research*, 3(7), pp. 331-333.

Mishra, D., n.d. SQL Injection Bypassing WAF. *OWASP*. Retrieved Feb. 28, 2021, from https://www.owasp.org/index.php/SQL_Injection_Bypassing_WAF.

Nguyen, T.H. et al, 2016. Towards a Science of Security Games. *In: Toni B. (eds) Mathematical Sciences with Multidisciplinary Applications*. Springer Proceedings in Mathematics & Statistics, 157. Springer, Cham.

Niculae, S., 2018. *Applying Reinforcement Learning and Genetic Algorithms in Game-Theoretic Cyber-Security*. Master Thesis.

Stock, A. et al, n.d. OWASP Top Ten. *OWASP*. Retrieved Feb. 28, 2021, from https://owasp.org/www-project-top-ten/.

Tripathi, J., Gautam, B. and Singh, S., 2018. Detection and Removal of XSS Vulnerabilities with the Help of Genetic Algorithm. *In International Journal of Applied Engineering Research*, 13(11), pp. 9835-9842.

Zheng, X. and Jin, J., 2012. Research for the application and safety of MD5 algorithm in password authentication. *9th International Conference on Fuzzy Systems and Knowledge Discovery*, pp. 2216-2219.