GitHub: https://github.com/pchiquini/CaseStudy.git

Security Assessment Report *WebGoat Inc*

Blind Numeric SQL Injection

A.T.O.M. Patrizio Chiquini & Omar Mahmud

Introduction:

WebGoat is an application that is used as a teaching tool for web application security. WebGoat provides a safe, legal space for people to test vulnerabilities and attacks in order to attain better knowledge on preventing and dealing with different types of vulnerabilities. During our assessment of WebGoat, we encountered the Blind Numeric SQL Injection vulnerability. Almost every web application employs a database to store the various kinds of information it needs to operate. The means of accessing information within the database is Structured Query Language (SQL), which can be used to read, update, add, and delete information. SQL is an interpreted language, and web applications commonly construct SQL statements that incorporate user-supplied data. Even though SQL work great and serves its purpose, it also opens a gap to web application for vulnerabilities such as SQL Injection, in this case our Blind Numeric SQL Injection.

The blind numeric version of SQL injection is when the hacker gets a 'true' or 'false' type of responses from the database that aid the attacker in exploiting a web application. In WebGoat, at first glance it is noticeable that the pin input field is the bottleneck of this vulnerability, where the hacker can keep asking the database if a number inputted by a client was greater than or less than a pin number associated with a particular account number and it would

keep responding with ‘true’ or ‘false’ (in this case ‘account valid’ or ‘invalid account’). Using a series of these statements a hacker is able to narrow down the inputted number to the valid pin number, thus getting your user's private pin number.

Finding:

The reason behind the success of the Blind numeric SQL injection on the WebGoat application is because in order to check the validity of the pin number the database is simply concatenating whatever data the user enters into a query statement in the database and checking if that statement is true. Thus, a user could concatenate and incorporate malicious text, including SQL code, in order to receive feedback from the application and execute the desired commands.

In WebGoat, a hacker could input a ‘SELECT’ SQL statement into the pin field and retrieve unrestricted sensitive data. The ‘SELECT’ SQL query causes the databases to check every row within the pintable and extract each record where the pin column has the value account and the pin column has the value we want. The field asking for a pin number is vulnerable since you can input these types of statements, which essentially communicate with the application and seek for a ‘true’ or ‘false’ type of response in order to find out whether the pin for a specific account number is greater than or less than a specified number. A hacker could use a series of these statements in order to ascertain the correct pin number for an account number he/she had access to using a simple “guess and check” method. With the correct pin number and account number a hacker is able to gain access to a user's private information that would be devastating for the company.

Mitigation:

There a couple of mitigation techniques you could implement into your application, however it is advisable to utilize parameterized queries. Parameterized queries would be an effective way to mitigate against SQL injection of any kind including the blind numeric version your application is dealing with. In parameterized queries, also known as prepared statements, the construction of a SQL statement, containing user input, is performed in two steps: the application specifies the query's structure, leaving placeholders for each item of user input, and the application specifies the contents of each placeholder. By using parameterized queries you have effectively separated the client input data from your database query which prevents your database from reading client input as malicious SQL code. The reason for this is that the query structure has already been defined, thus the relevant API handles any type of placeholder data in a safe manner, so it is always interpreted as data rather than part of the statement's structure.

Another possible mitigation specifically for the blind numeric version of SQL injection would be to lock your input field for vary lengths of time after a certain number of incorrect inputs of a pin number. For example, if there are 5 incorrect guesses of the pin lock the input field and prevent the client from inputting any more data for a minute, then if they input incorrectly again increase the lock time to 5 minutes, then 15 minutes, then an hour and so on. By doing this you make the hacker have to do an incredible amount of work to find the correct customer pin numbers since they need to input many incorrect SQL strings to find the correct one. This mitigation technique will not eliminate the SQL injection vulnerability from your application; however, it will make finding a customer pin much harder and challenging for a hacker that would deter many from trying.

Lastly, another simple mitigation technique would be to control the number of characters that can go into the text field where the user is inserting allowed to insert their pin number. By

adding this simple restriction, you will be able to control and manage the type of input that it is placed in your database query.

Automation:

While exploiting our vulnerability we were able to notice some similarities from using a regular web browser and creating a Gauntlt test case. When we initially started testing the browser manually we were sure to have ZAP open catching every response that the site provided for us in order to identify and track the browser's behavior. In particular, we paid close attention to the login page request and to the request sent from our vulnerability attack page. Once we had reached our vulnerabilities page we manually started inserting SQL statements and narrowing down our PIN number with a simple algorithm testing for upper and lower bounds, which, after some time, led us to our answer. In contrast, our automated attack accesses the vulnerability page and started posting the test as soon as it navigated to the page with the use of the Python's HTTP responses library. One of the conditions that made our automation attack unique was the fact that it had to deal with parsing through the JSON that the page sent out after every attempt that was made while exploiting the vulnerability. We had to incorporate some conditional statements that allow our algorithm to analyze the JSON received before inserting and testing another number. Even though it is an additional step our automated attack had to deal with, it was still an efficient way to solve and exploit the vulnerability

Test Harness:

In order to test and verify if the vulnerability is still present we recommend for you to run our Gauntlt attack file, which will inform you the status of the vulnerability present. You will know the vulnerability is still present if Gauntlt still fails when you run it. If the Gauntlt attack

file fails when you run the attack, it will indicate that our code still was able to use SQL statements to reach and find the customer's pin number. In other words, our Gauntlt file is written to pass the test with the text "vulnerability not present" when the vulnerability has been fully fixed, which you will seen with green text. If you fix the vulnerability when our code attempts to input an SQL statement into your pin field an error message should be returned in the http response which would cause our code to immediately return and print "vuln-21 is NOT present," which would cause Gauntlt to pass. To the hacker, an http response that returns an error indicates that when he/she tried to input the SQL statement all that was returned was an error message which gives no information as to whether or not the correct customer pin is greater than or less than the hackers inputted number thus blind numeric SQL injection cannot occur. As of right now, when you run the Gauntlt file you will see the red statement "vuln-21 is present. Attack Successful" which means that our algorithm is still able to successfully inject SQL and acquire the desired pin number.

Conclusion:

As stated previously, you will know the vulnerability is fixed by running our Gauntlt attack file, which will pass once the vulnerability has been cleared and properly handled. The test will only pass when our algorithm is unable to utilize the SQL statements to analyze and differentiate an account from being invalid and valid causing our algorithm to print "vuln-21 is NOT present" and exit. Furthermore, our algorithm that simulates this type of SQL injection allows us to narrow down the search by using the site's own text responses against it, such as "invalid account number" or "account number valid". By handling this type of vulnerability you will avoid an attack from injecting into the web application data where he may be able to read and modify sensitive application data, gather pin numbers from different accounts, or perform

other unauthorized actions. After reading on this report, we hope you are aware of the vulnerability you are currently presented with and we hope that we have informed you about the risks of this attack. As stated previously, we advise you to implement and set-up parameterized queries for your pin field, as that would be the best mitigation technique for Blind Numeric SQL Injection.