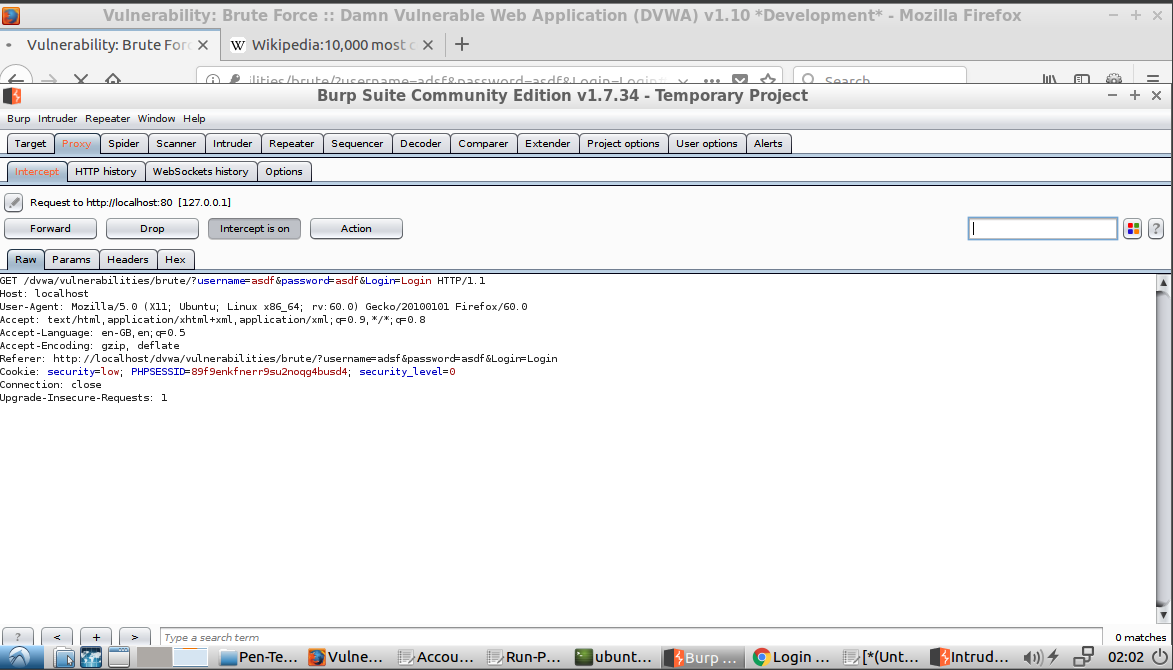
# Group Project 2

(Group 1)

Mario Amaro  
Sushruti Bansod  
Joshua Fishman  
Nick Forleo

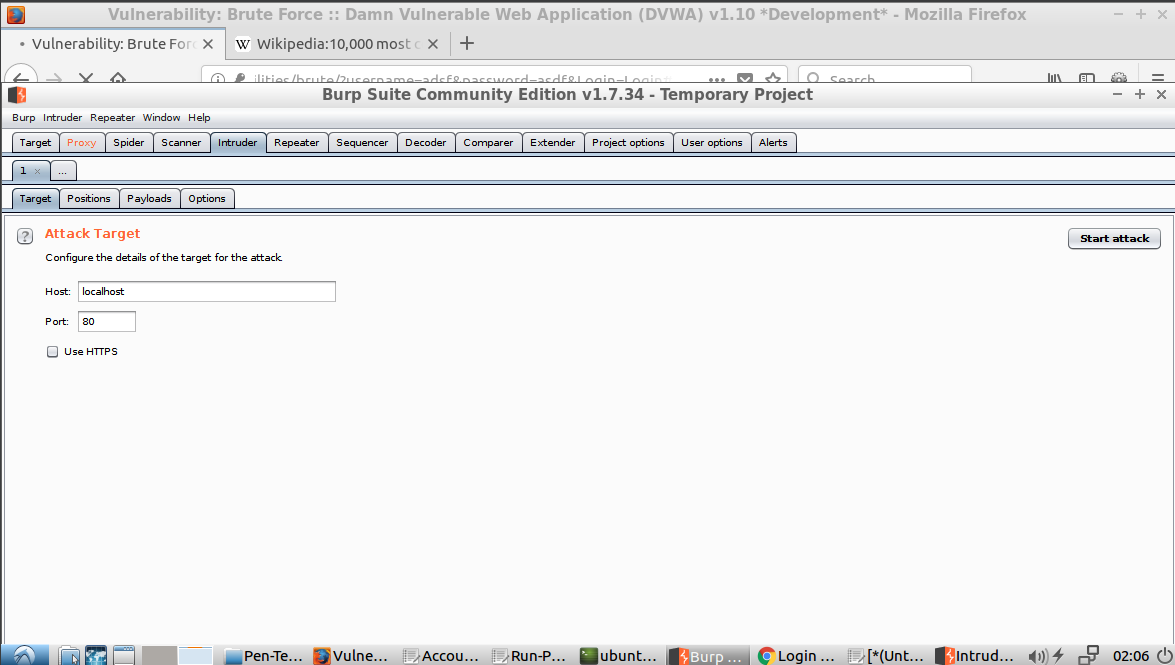
**Task 1.1:** Scenario 1 of attacking authentication

For this scenario, we exploited an authentication vulnerability in DVWA. We used BurpSuite to monitor the conversation between the host and the client. In this authentication attack, we looked at a brute force type of attack to try and guess the username and password. We were able identify a weak authentication policy that allowed us to try unlimited username and password combinations. First, we needed to identify the necessary form field required to login. To do this, we intercepted a login attempt request and noted the login parameters.



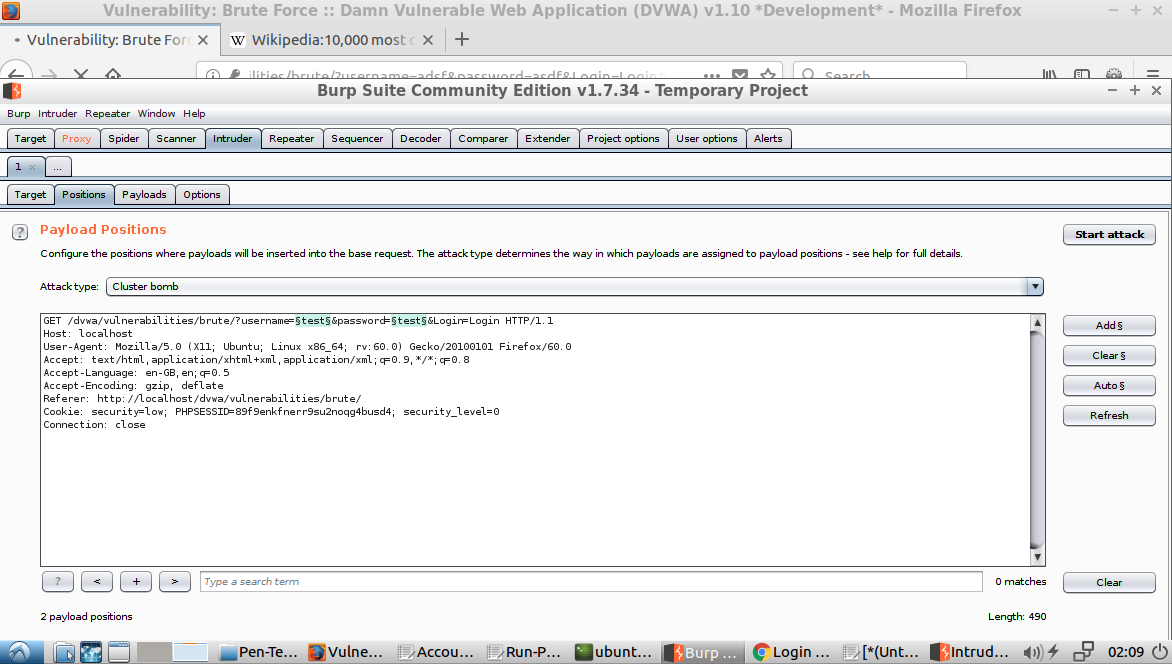
*Figure 1.0 - Intercepted login request*

In figure 1.0, you can see that it is looking for the fields ‘username’ and ‘password’. I then compiled a list of the most common usernames and passwords. This list was used to execute a cluster attack against the login page. This was also done using Burp Suite. The ‘Intruder’ tab allows you execture specific types of attacks. First you need to define the target, as shown in figure 1.1.



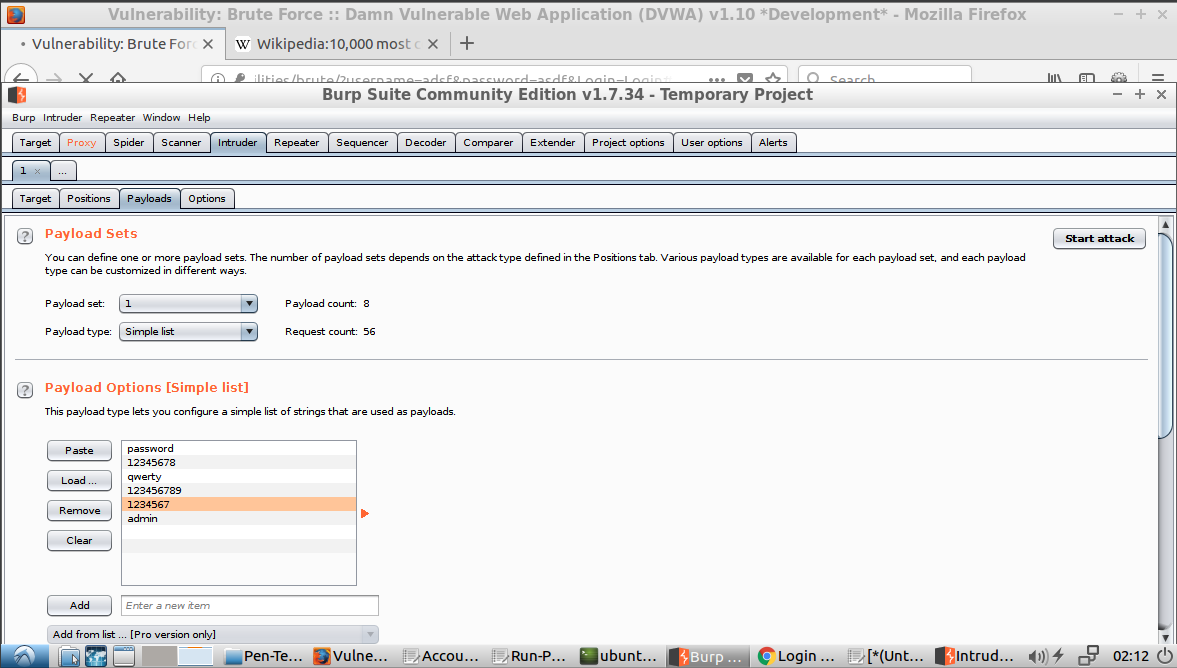
*Figure 1.2 - Define Target in Intruder*

This is targeted DVWA since it is locally hosted on port 80. The next step is to define the Attack Type and request. We used a Cluster bomb attack. For the request, we abstracted out the values for username and password so that Burp Suite would know where to inject the values.

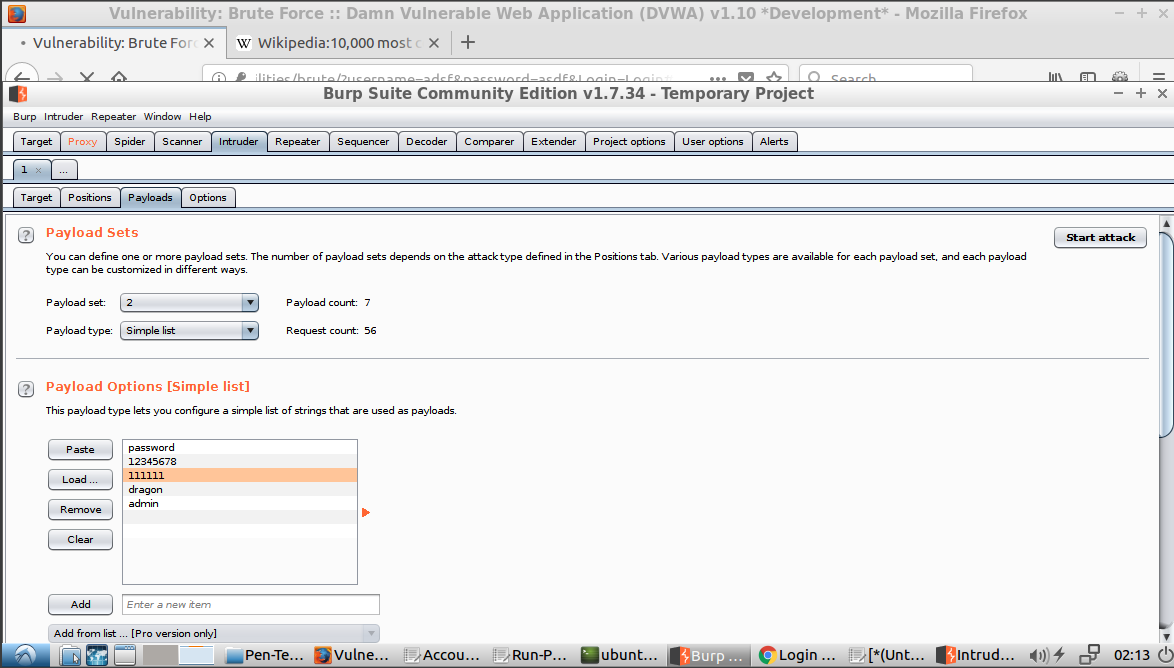


*Figure 1.3 - Define attack type and abstracted request*

The next step is then define the value to be injected into the request. We used the most common usernames and password mentioned above. We defined payload set 1 as the usernames and payload set 2 as the passwords. Figure 1.4 shows the values of the username set that were injected and figure 1.5 shows the passwords.

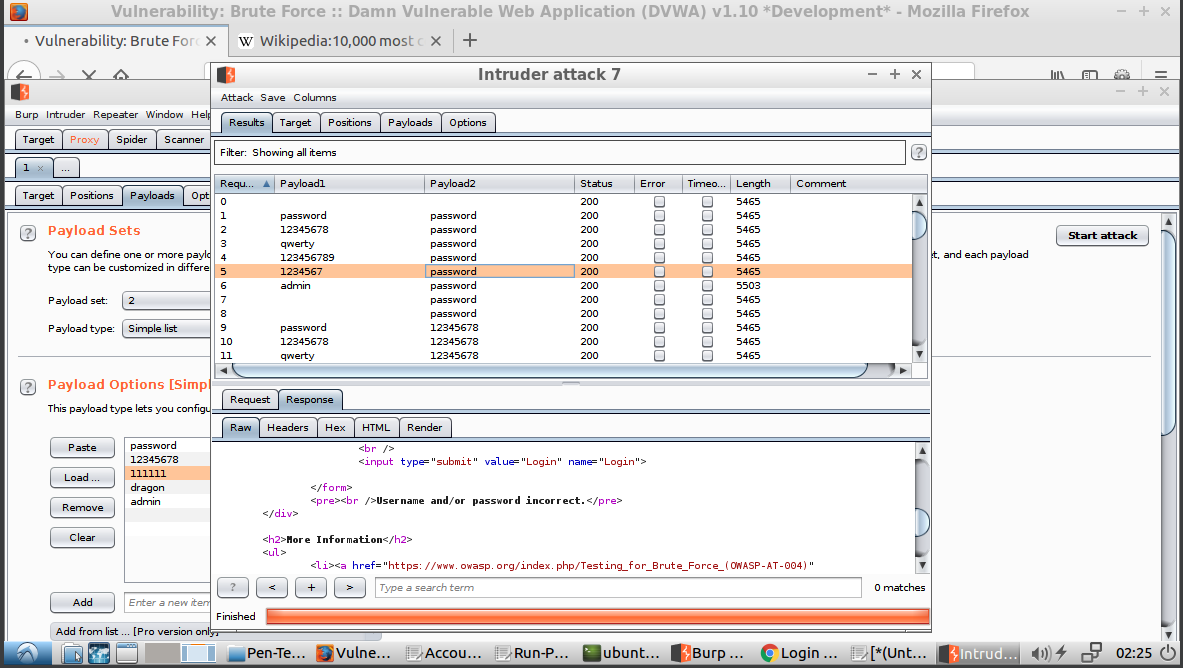


*Figure 1.4 - Username Payload Set*

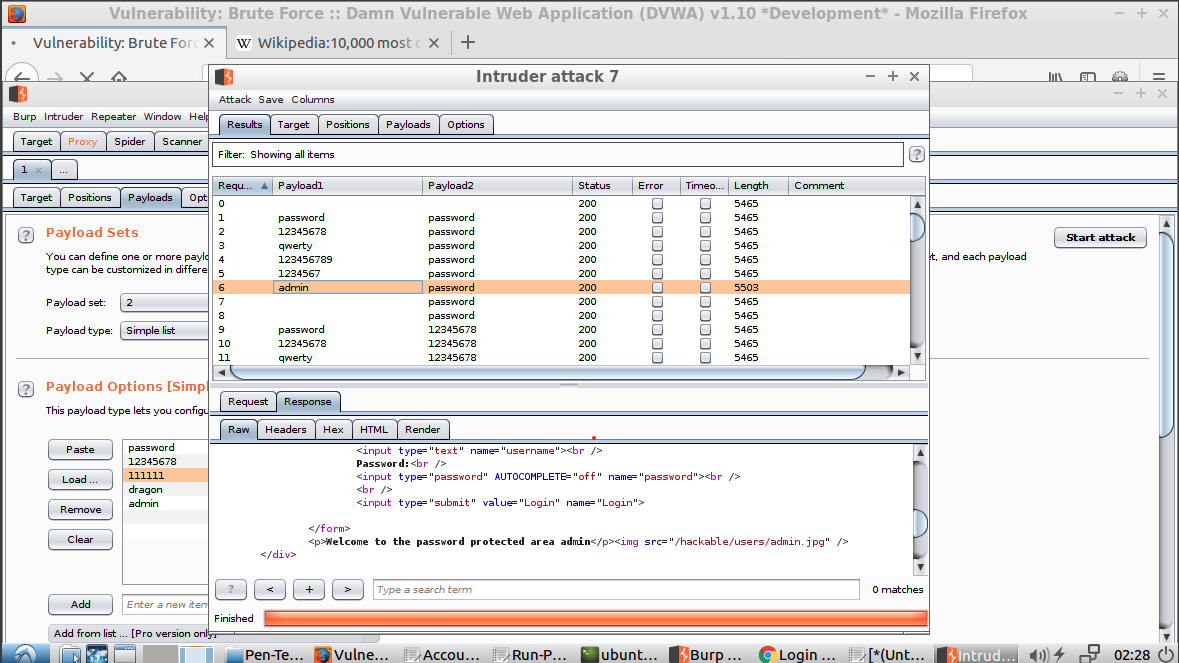


*Figure 1.5 - Password Payload set*

The attack is now ready to be executed. It will execute the define request while injecting all possible combinations of each payload set into the request. We were then able to go through each request and inspect the response. If the username and password was wrong, the response will indicate the username/password combination was incorrect, but will also tell us if the combination was correct. For example, if we looked at the request shown in figure 1.6, you can see that the wrong combination was injected because the response include “Username and/or password incorrect.” However, figure 1.7 will contain “Welcome to the password protected area admin.” This means the attack was able to correctly identify the correct login combination, which was admin:password.



*Figure 1.6 - Incorrect combination parameters and response from cluster attack*

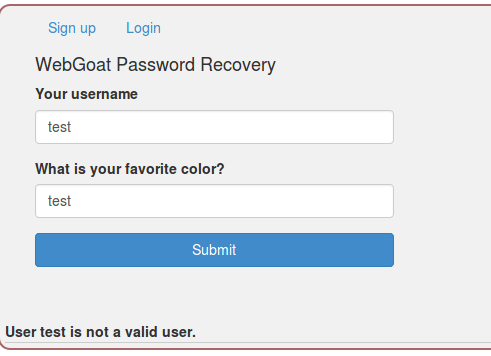


*Figure 1.7 - Correct login information found via cluster bomb attack*

The success of this attack highlighted 2 issues with the authenication of this application. The first being the aforemention policy of allowing repeated attempted logins. The second is that the application does not enforce any stronger password polices (i.e. must have a number, must have a capital letter, etc...), since the correct login information was easily guessed.

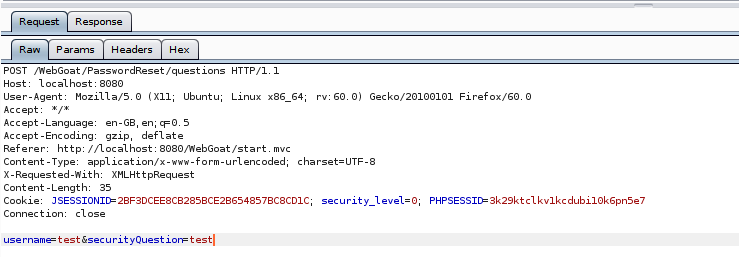
**Task 1.2:** Scenario 2 of attacking authentication

For this scenario, we exploited an authentication vulnerability in WebGoat. On the “Authentication Flaws > Password reset” page, the server provides verbose authentication failure messages during the password reset process. Using Burp Suite running on port 8000 (so it doesn’t conflict with port 8080 which runs WebGoat), the attacker can visit this page in a web browser as seen in figure XX to submit a password recovery request.



*Figure 1.8 - Example password recovery interface with invalid user.*

This particular example used an invalid user which can be seen in the text at the bottom of figure 1.8. The HTTP request and response can be viewed in Burp Suite as shown in figures 1.9 and 1.10.

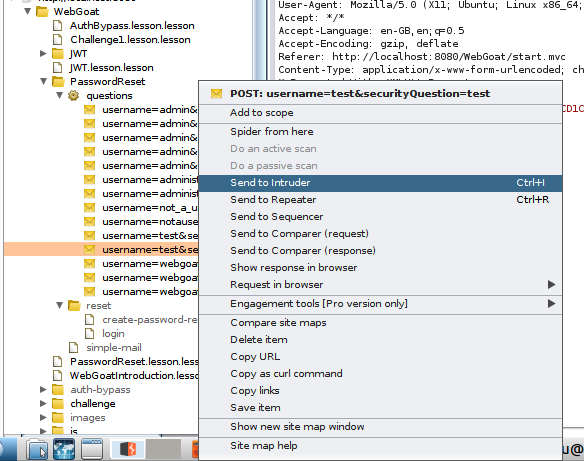


*Figure 1.9 - Example password recovery request in Burp Suite*



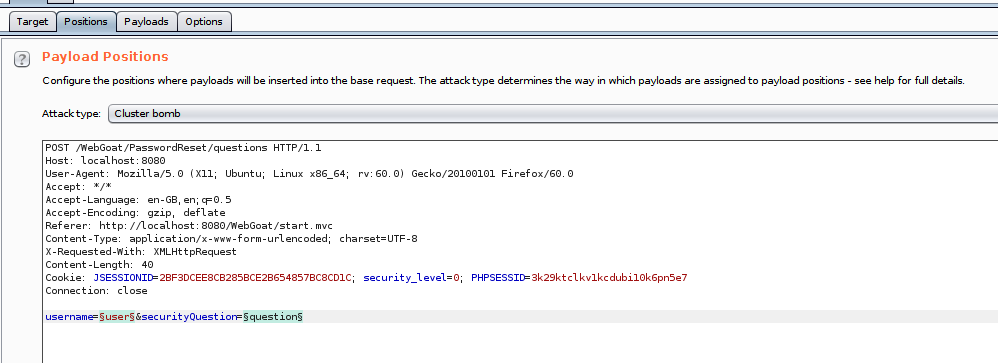
*Figure 1.10 - Example password recovery response in Burp Suite.*

Given that their are two parameters needed to reset a password, we used the technique described in Task 1.1 to setup an Intruder attack. The first step was to send the request to intruder.



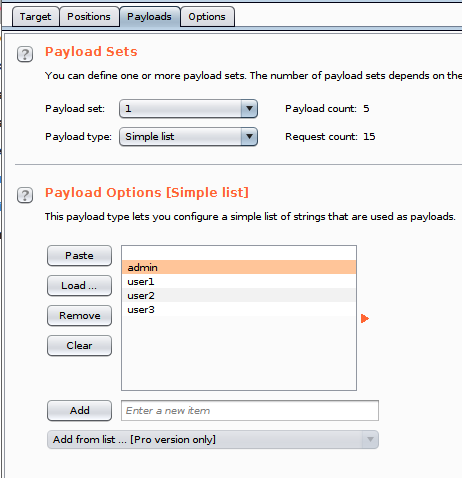
*Figure 1.11 - Send existing request to Intruder in Burp Suite.*

Once the request was loaded into intruder, the values of username and securityQuestion were setup as the two payload positions for a cluster bomb attack.



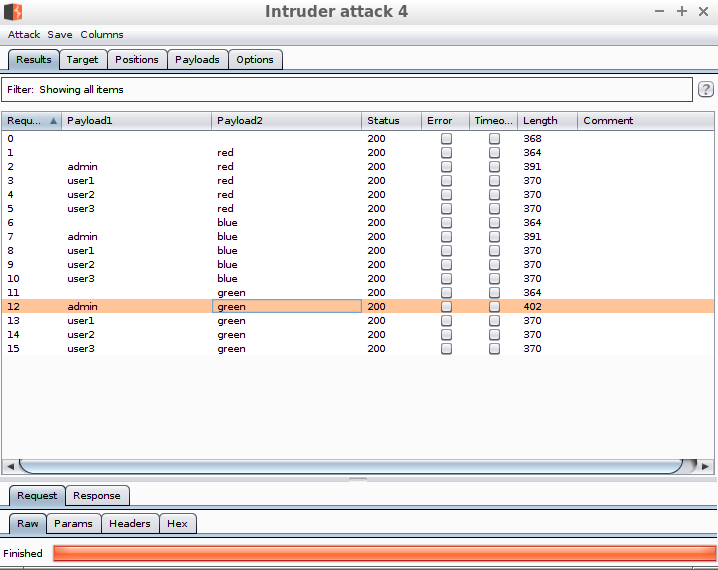
*Figure 1.12 - Burp Suite Intruder cluster bomb payload position setup.*

Given the two payloads called “user” and “question” have certain values that are more likely than others, two different payload sets could be created to do a brute force attack. In figure 1.13, the values for user are admin, user1, user2, and user3 for payload set 1. Payload set 2, the answer to the question, has the value red, blue, and green.



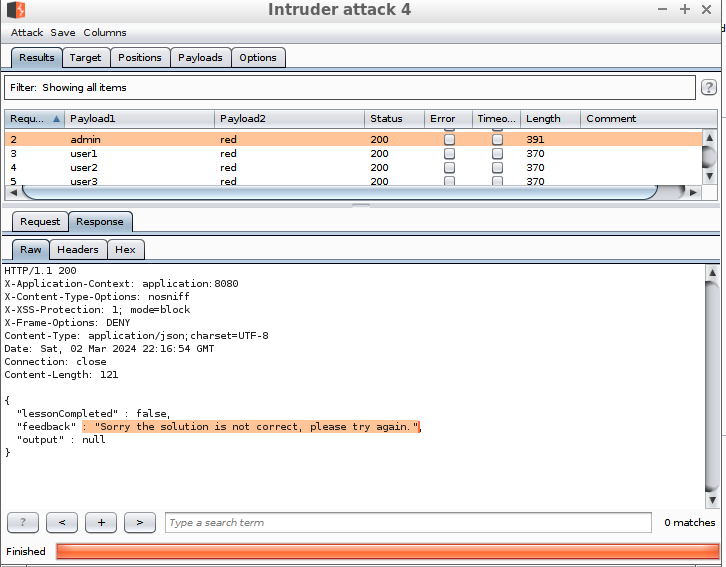
*Figure 1.13 - Setting up Burp Suite Intruder payload sample list for cluster bomb attack.*

After this configuration, we triggered the attack and evaluated the results as seen in figure 1.14.



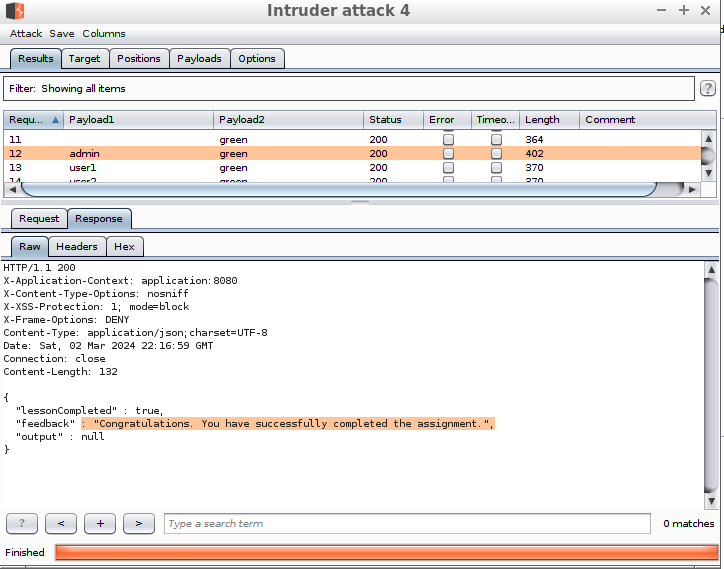
*Figure 1.14 - Intruder cluster bomb attack results.*

As shown in the results, the file length of the response message was always longer with the “admin” user. Looking more closely at one of those responses in figure 1.15, we see that the server response tells us that “admin” is a valid account, becuase the error message is different from the typical unsuccessful response in figure 1.8.



*Figure 1.15 - Admin user appears to be a valid user.*

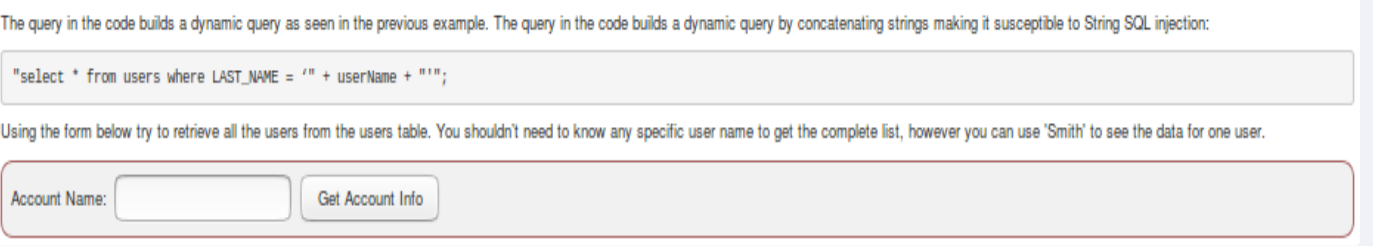
To optimize the cluster bomb attack, we could then only target legitimate users and a list of common favorite colors, but fortunately, this was unnecessary, becuase our first attack uncovered a valid combination of user and security question answer. admin's favorite color was “green”, and we can use this information to reset admin’s password. Figure 1.14 showed that this combination had the largest file length which hinted that this was the best combination. Looking at the body of the response shown in figure 1.16 confirms this.



*Figure 1.16. Successful validation that “admin” has the favorite color of “green”.*

**Task 2.1:** Scenario 1 of attacking authorization

This attack was done using Webgoat and WebScarab. The challenge was to view all the data in the table with multiple users and credit card information.



*Figure 2.1 – WebGoat form for account information*

The first step was having the intercept request turned on.

A screenshot of a computer

Description automatically generated  
*Figure 2.2 – WebScarab Proxy settings*

When the form was submitted, WebScarab intercepted the request. From the data there was no encryption set.

A screenshot of a computer

Description automatically generated

*Figure 2.3 – WebScarab request intecept*

We injected a basic SQL string to see if could pull all the data out.

A screenshot of a computer

Description automatically generated

*Figure 2.4 – WebScarab Request intercept with changes*

The SQL injection was successful, and we were able to view all the data from the table.

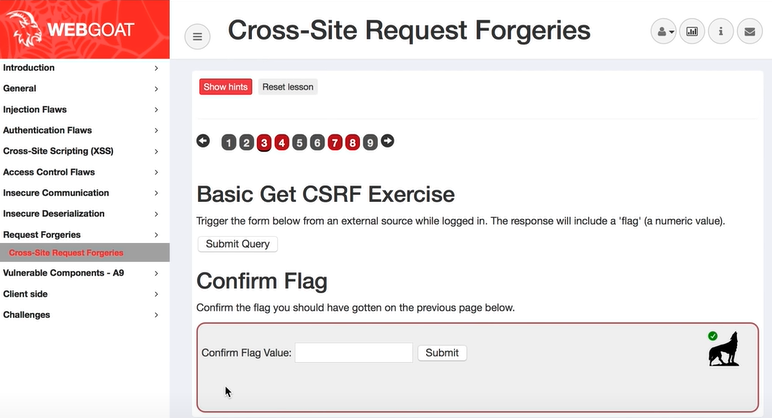
A screenshot of a computer

Description automatically generated

*Figure 2.5 – WebGoat Credit Card Data*

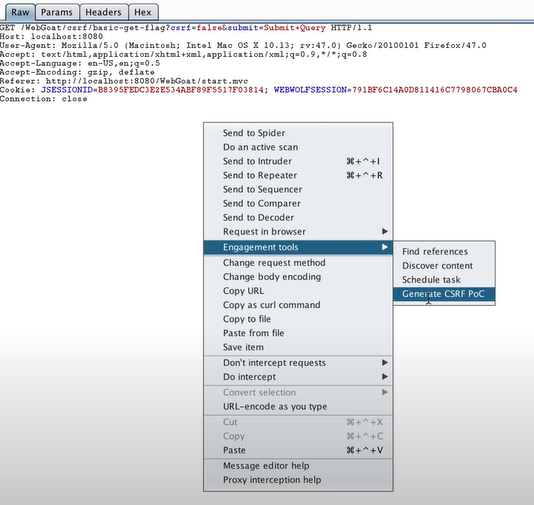
**Task 2.2:** Scenario 2 of attacking authorization

To compromise the authorization of a website I chose to exploit one of the vulnerabilities in WebGoat using Burp Suite. Here I will be trying to solve the Cross-Site Request Forgeries.



*Figure 2.6 - CSRF*

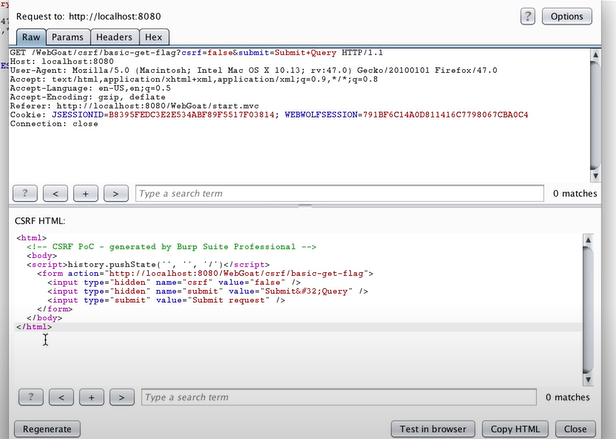
First, I wanted to check the type of request that was being sent when the submit query button was clicked on the website. I can see that this is a GET request for the server. So as an attacker, I needed to create this get request externally. Below are the results:



*Figure 2.7 - Generating CSRF PoC*

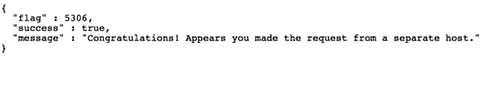
Burp suite has many interesting feautes. One of them is the CSRF PoC (proof of concept). This allows us to see how the request is being generated.

Here it generates some basic HTML code for a button that creates the get request.



*Figure 2.8 - Generated HTML Code*

Next I saved this html file and opened it in the browser. When I clicked on the button, i could see the flag and the success message.



*Figure 2.8 - flag and success message.*

**Task 3:** Based on the found vulnerable issues, what should you do to prevent attackers from hacking your web application?

**Task 3.1:** Protection/prevention for Scenario 1 of attacking authentication

There are plenty of steps that can be taken to prevent and/or protect against the attack method used in scenario 1. A combination of the following actions would be most effective. They are not listed in any particular order. The first idea that we came up with was to limit the number of consecutive, unsuccessful logins, such as 5. This means there are only 5 possible attempts that can be executed per guessed credential before the account is locked for a specified period of time.

A second idea is that each login attempt is sent along with a unique identifier that the server is already aware of. For example, when the server receives the request to load the login page, it also generates a checksum value that it expects to receive back with the login request. If the login is unsuccessful, it generates a new key and sends it back with the error message. The cycle then repeats until there is a successful login. This would prevent a bad actor from attempting to repeatedly send requests with different login values, like the method we used to attack the authentication.

A third idea is to enforce stricter password policies, such as including special characters, capital letters, numbers, etc.. This would prevent someone from maintaining a list of commonly used passwords that can be used to brute force their way into an application.

**Task 3.2:** Protection/prevention for Scenario 2 of attacking authentication

To protect against authentication scenario 2, all the concepts from scenario 1 apply. The only exception is that in this scenario, there were no passwords, but there was a security question instead. In scenario 1, common passwords created a problem, but in scenario 2, the problem is that the security question prompted many users to use common answers such as “red”, “blue”, and “green”. These answers are easy to crack and don’t require knowing the user to guess them. Instead, security questions should be chosen so that the user provides answers that are unique to them and not common among many people. Alternatively, recovery phone numbers or email addresses are a safer approach to resetting passwords that don’t give the attacker an opportunity to reset the password on a user’s behalf.

Furthermore, scenario 2 has another problem that scenario 1 doesn’t have: verbose authentication failure messages. If an attacker submits multiple password reset requests, each with a different potential username, the verbose response will tell the attacker which of the usernames are legitimate users. Then, they can use the condensed list of valid usernames to test possible answers to the security question for each user, drastically reducing the space of valid combinations. To avoid this issue, the server should not provide extra infromation to the client if the username was a valid username. If either the username is invalid or the security question is incorrect, the same error message should be used, such as “invalid user or security question, please try again”.

**Task 3.3:** Protection/prevention for Scenario 1 of attacking authorization

To protect against SQL injections developers should use prepared statements, stored procedures, input validation, and authorization roles.

Use prepared statements to separate code and data to ensure that the database treats user input literally, even if its malicious. Developers will define the SQL code first, then pass individual parameters later, preventing attackers from altering the query's intent.

Stored procedures are pre-defined SQL code snippets that reside in the database and accept parameters. They offer similar protection when implemented safely, however poor use of these can be exploited by attackers.

Input validation and authorization roles will provide better security by not allowing everyone access to everything in the database.

**Task 3.4:** Protection/**prevention** for Scenario 2 of attacking authorization

To safeguard against Cross-Site Request Forgery (CSRF) attacks, where attackers exploit the trust a web application has in a user's browser, adopting secure coding practices and implementing robust security measures are essential. One foundational step is to shift from using GET to POST requests for all state-changing operations within the application. GET requests can be easily forged, making them a less secure choice for actions that alter data or state on the server. Moreover, the inclusion of anti-CSRF tokens in web forms acts as a critical defense mechanism. These tokens, unique to each user session, verify that requests sent to the server originate from the application's own forms, thereby preventing unauthorized requests from being processed.

Web development frameworks often come equipped with built-in CSRF protections, which automatically handle the generation and validation of CSRF tokens. Utilizing these features to their full potential can significantly streamline the process of securing applications against CSRF attacks. The validation of `Referer` and `Origin` HTTP headers serves as an additional security measure, helping to confirm that requests are initiated from trusted sources, despite potential limitations due to header spoofing and the occasional absence of these headers.

Through the collective implementation of these strategies, organizations can achieve a robust defense against CSRF attacks, safeguarding both their applications and users from unauthorized actions.

**Team peer-review table**

|  |  |
| --- | --- |
| Name | Contributing Efforts in this project (0 ~ 100%) |
| Nick Forleo | 100% |
| Mario Amaro | 100% |
| Sushruti Bansod | 100 % |
| Joshua Fishman | 100% |

Optional **Appendix** (if any)

1.1: Partial list of commonly used passwords/usernames sources from: https://en.wikipedia.org/wiki/Wikipedia:10,000\_most\_common\_passwords