

# Final Report

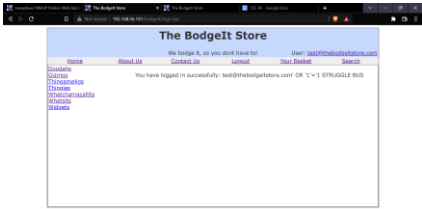
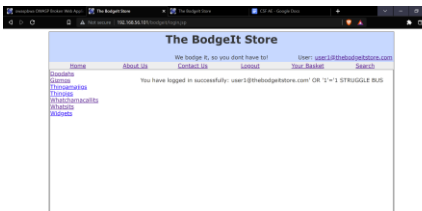
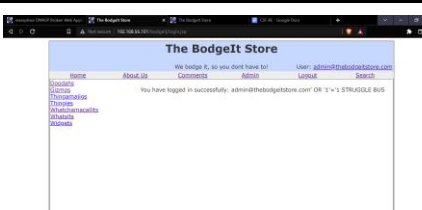
by

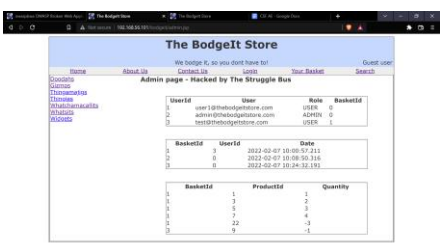
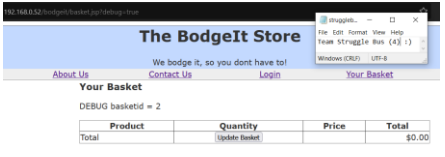
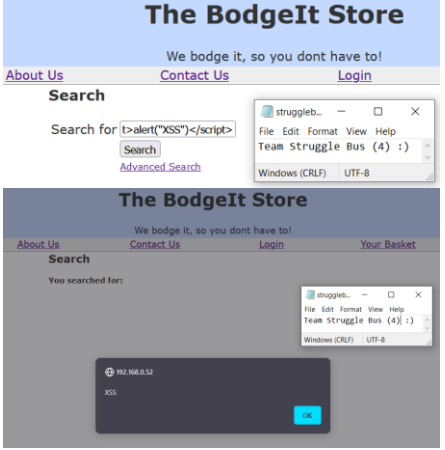
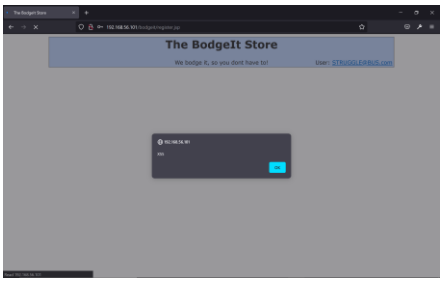
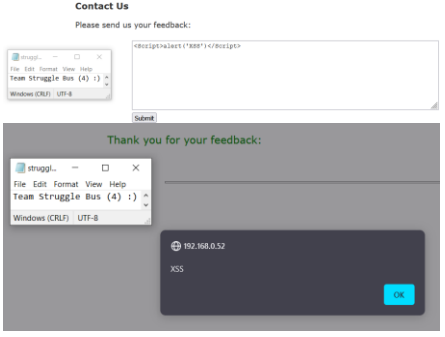
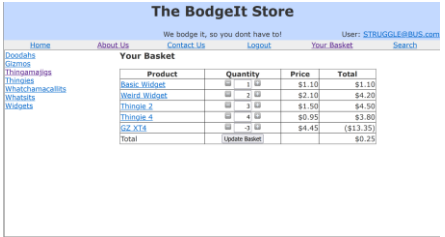
## Team Struggle Bus

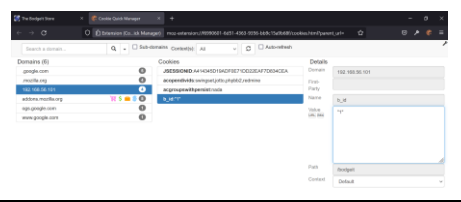

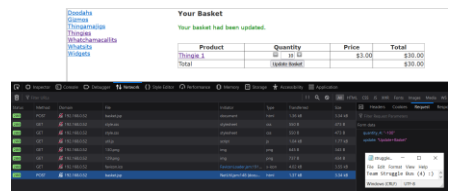
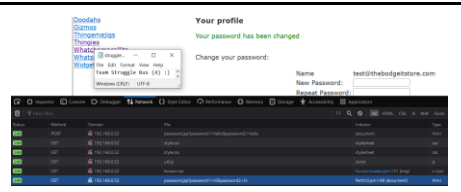
Donald MacKenzie(246230M) Kārlis Siders (2467273S) Khenā Dungu(2462068D) Declan  
McBride(2399448M) Rishabh Mathur(2465899M) Lau Hok Yee(2551157L)

### Attack Report

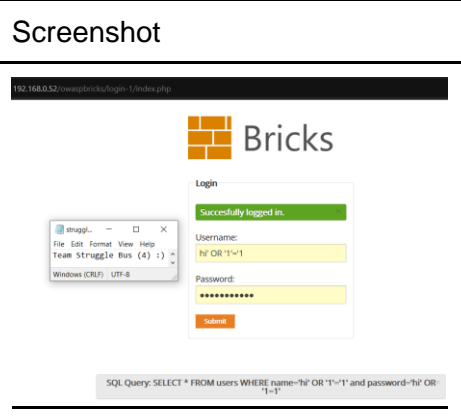
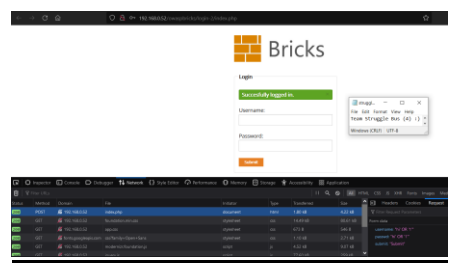

#### The BodgeIt Store

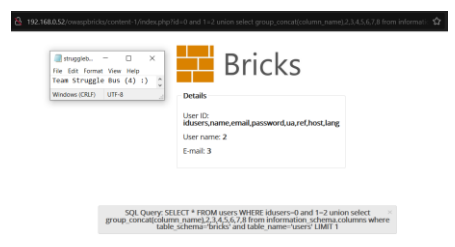
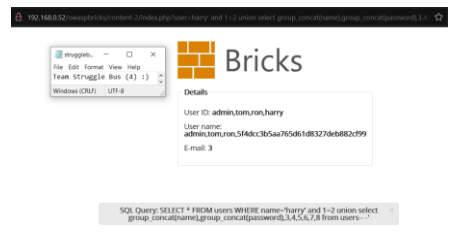
No.	Type of attack	Description	Screenshot
1.	SQL Injection	Logged in as another user ( <a href="mailto:test@thebodgeitstore.com">test@thebodgeitstore.com</a> ) by injecting SQL in log-in page	
2.	SQL Injection	Same as above: <a href="mailto:user1@thebodgeitstore.com">user1@thebodgeitstore.com</a>	
3.	SQL Injection	Same as above: <a href="mailto:admin@thebodgeitstore.com">admin@thebodgeitstore.com</a>	

4.	Broken Access Control	Found hidden content (an admin URL) as a non-admin user. Detected as a comment left in the HTML source of the home page.	
5.	Broken Access Control	Found debug information by exploiting open GET request in URL	
6.	Cross-Site Scripting	Displayed JS popup in Search page.	
7.	Cross-Site Scripting	Displayed JS popup when registering (in password field).	
8.	Cross-Site Scripting	Displayed JS popup in Contact Us form with slight modifications to the popup script.	
9.	Broken Access Control	Accessed someone else's basket.	

			
10.	Web Parameter Tampering	Forced someone to add an item to their basket with a GET request.	
11.	Web Parameter Tampering	Got the store to owe money to the user.	
12.	Web Parameter Tampering	Changed password with GET request.	

### OWASP Bricks

No.	Type of Attack	Description	Screenshot
13	SQL Injection	Login 1: Logged in as another user with SQL query.	
14.	SQL Injection & Web Parameter Tampering	Login 2: Logged in as another user by directly injecting SQL in GET request.	
15.	Cryptographic Failures	Content 6: Discovered host details by finding out encryption used (base64) and encrypting SQL injection to this.	

16.	Web Parameter Tampering	Content 1: Discovered database table structure for users by exploiting URL-visible GET request.	
17.	Web Parameter Tampering	Content 2: Found all usernames and passwords by exploiting GET request in URL.	

## Defence Report

### Injectons (SQL and Cross-Site Scripting (XSS))

We found that both BodgeIt and OWASP Bricks were vulnerable to injection attacks. SQL and XSS attacks applied to both web apps. SQL injections involve using queries to the database to gain access to hidden information without authorisation. XSS injections involve an attacker injecting malicious scripts into a web app. These attacks mostly occur when characters that are typically used to differentiate between commands and the data itself are allowed to be passed directly to the interpreter.

There are various methods that may be applied to prevent SQL injection attacks. Techniques include validating user inputs, thereby ensuring that all the inputs made by the user are legitimate. This also grants us the ability to filter out any unexpected or invalid inputs. While designing, it is essential to ensure that the parameters are added as part of the final query string. This prevents commands being altered maliciously. Queries with parameters allow for pre-compiling SQL statements to ensure that the database is able to distinguish between code and user input data. By pre-compiling SQL commands that a web app will use multiple times, we can reduce the number of opportunities attackers will have to make SQL injection attacks.

Another important tool that can be used to reduce a web app's vulnerability to injection attacks is to consciously give users the database privileges they need and no more. For example, a user may need the ability to write to a relation and query a relation, but perhaps only superusers may need the ability to remove from a relation. A philosophy of granting different users only the access privileges they need and no more can be a potent tool against attacks in and of itself.

To prevent XSS attacks in this given scenario, it is essential to encode output data as well as input data, particularly where user-controllable data is an output delivered in HTTP responses. At the point where user-controllable data is output in HTTP responses, encoding the output to prevent it from being interpreted as active content reduces the vulnerability to the XSS attack significantly.

## Broken Access Control

The URL that was accessed during the Attack stage, /admin.jsp (#4), should have been checking user access privileges while on the page, not just when navigating to it.

Furthermore, website HTML and CSS files should be cleaned of unnecessary (and sometimes dangerously transparent) comments left over from the development stage. These weaknesses are common because of ineffective functional testing during the development phase.

Manually testing is one of the most effective ways to detect broken access control. This is done by including the HTTP method, and the use of GET, PUT, controllers, direct object references, etc in our testing.

Another way to prepare against broken access control is to ensure that policies for assigning necessary permissions are thorough. They should only grant access for editing to the relevant stakeholders. Like with injection attacks, following a policy of least privilege and only granting personnel who need to edit certain files write access is best practice for reducing a web app's risk of attack.

On restricted pages client-side caching should be disabled as it may allow others to re-access those sites. Quite often web applications like the ones used in the attack stage are accessed from public access points such as schools, libraries, or from a place where a user may find free to use Internet access. Browsers quite frequently have a tendency to cache web pages; therefore, they may be accessible to attackers. Attackers then gain unauthorised access to different parts of the website, just the way the team was able to get access to admin.jsp. To prevent caching done by the browsers, developers should use techniques like implementing HTTP headers and meta tags. This is particularly important in pages that contain sensitive information and might be particularly attractive to attackers.

Most web applications have servers that rely heavily on accessing the control lists that are provided by the file system of the platform. This means that, even if all of the data is stored on the backend servers, quite often there would be files stored locally on the web and on the application server. Such files should not be accessible. For example, in this situation the file admin.jsp was quite easily accessible. Ease of accessibility makes the attack a lot easier, and it is essential to ensure that the files that are to be presented to the users are marked readable using the operating software's permissions. Additionally, most of the files should not only be non-readable but should also be non-executable. To avoid the attacker misusing the URL, it is necessary to escape user entries in order to avoid path traversal patterns (such as "../" for example).

## Cryptographic Failures

Some researchers have found that data that is transmitted in clear text using protocols such as HTTP, SMTP, FTP, and protocols using TLS upgrades like STARTTLS can prove to be extremely hazardous for the web app.

Using old or weak cryptographic algorithms or protocols used either by default or in older code and using or generating Default crypto keys and weak crypto keys are also some of the other potential concerns that can be linked to the web apps facing cryptographic failures.

Even when the broken app, OWASP Bricks, does try to obfuscate data intended to be hidden, a weak encryption algorithm is used - base64 (#15). Encoding in base64 gives an opportunity for any malicious party to easily decode the base64 string with online-accessible resources and access hidden content, especially if it hides input fields which can be used in conjunction with SQL Injections, which are easy to encode in base64 as well. In order to avoid this, a better encryption algorithm should be used, such as RSA encryption.

It is of prime importance to classify all data, and define whether it is processed, stored, or transmitted by an application, in order to give a clear understanding of what data needs to be defended, and use the appropriate encryption techniques to store it. It is important to use encryption wherever it is possible to ensure that data can't be decrypted even if the session keys are exposed.

Another way to ensure that the cryptographic failures are defending against in a robust way is to ensure that all data is encrypted in transit with secure protocols such as TLS with forward secrecy (FS) ciphers, cipher prioritisation by the server, and secure parameters. Also, enforcing encryption using directives like HTTP Strict Transport Security (HSTS) would certainly help prevent cases of cryptographic failure because, like those seen in the OWASP Bricks app (which usually uses the obfuscation technique). If a strong and robust algorithm is used along with the right protocol techniques, the risk of such attacks is reduced quite significantly. Disabling caching for responses that may contain sensitive data and not using legacy protocols such as FTP and SMTP for transporting sensitive data, are some of the other mechanisms that could be used to bolster the overall webapp.

## Web Parameter Tampering

By allowing to directly edit GET and POST requests and their parameters, the Bodgeit Store and the OWASP Bricks app are both vulnerable to web parameter tampering attacks.

Malicious attackers are able to avoid any logic and sanitisation mechanisms in place for direct user input and can, for example, get a vulnerable store to owe them money or access other users' data.

It is very common for web apps to do client side validation using Javascript before sending requests over to the server. By tampering with the parameters of the POST Variables, data that is invalid can be easily sent by potential attackers to the server. The server then accepts and processes this data as if it is valid. In general, parameter tampering in these cases can be stopped by ensuring that both the client and the server use firewalls for protection, and that all the session cookies that are being used are encrypted using robust encryption algorithms.

In addition, it is very important to ensure that there are no parameters that are directly used in the query string, and that the forms on the webapp use regular expressions to validate and reduce the data used in forms. It is essential to always verify the user input, as invalid input is the primary reason for these types of attacks. It is critical that a validation procedure is in place on the server side, as it is very easy to bypass client side validation.

The BodgeIt Store and OWASP Bricks apps could also prevent these types of attacks by implementing checking mechanisms directly for web requests, like checking user privileges for access and establishing boundaries after testing extreme (edge) cases, e.g., negative items in a basket.

Finally, encrypting session cookies can prevent tampering, as can implementing a policy for preventing client-side generated cookies from making any security related decisions.

## Conclusions

In conclusion, there's no silver bullet for web application security as these threats continue to grow and evolve, however, it is possible to avoid web attacks of these kinds to a great extent if the necessary security features are identified during the development, coding and testing phase and are considered through-out the entire development life cycle. A wide variety of attacks were identified during the attack phase, both in the Bodgeit Store webapp and in the OWASP Bricks webapp. Injection attacks, Broken Access Control attacks, Cryptographic failure attacks, and Web Parameter Tampering attacks were the most common. This report has laid out a variety of precautions that may be taken in order to secure against such attacks in the defence phase. The primary recurring themes are strict user access policies, such as implementing a least access policy for database access, and ensuring that the most up to date cryptography algorithms are used for encoding and decoding user data. Other preventative methods were noted, including implementing client and server side firewalls, and preventing parameters being entered into a query string. Our team worked well together as a whole and managed to communicate any struggles effectively through our Teams chat. The process of figuring out how to complete the web parameter tampering for the Bodgeit Store was a source of frustration amongst the team. A suitable solution was reached after discussing it together in the labs.

## Group Policy

Karlis: Did 5 OWASP Bricks attacks, 6 The BodgeIt Store attacks, started 4 paragraphs in Defence Report.

Donald:

Khena: Added to and tidied the Injections section. Proofreading and tidying of the entire report

Declan: Added detail to the Defence Report (Injection attack section, Web parameter testing).

Proofreading and tidying of the entire report. Wrote the conclusion to report.

Rishabh: Analysed the attacks in the defence report.

Lau Hok Yee (Natalie): N/A