**INF226 - Eksamen Vår 2019**

**Exercise 1:**

1. False. It can be caused by other types of code, not just JavaScript. It happens when third party users/sites inject code into our program, which is then executed.
2. True
3. False. The length of the queries doesn’t matter, how they are built matters.
4. True. Memory safe languages prevent buffer overflow.
5. True
6. True
7. True
8. True
9. True. They are also prevented with prepared statements.
10. True
11. False. They should not take up as little as possible because that would make it easier to brute-force it
12. False. Can only passed from using request on the with the same origin. (http -> http, but not http -> https) ??
13. True
14. True
15. True
16. True
17. False.
18. True
19. False
20. False

**Exercise 2:**



Availability: Available to all users when needed.

Confidentiality: Data are protected and hidden from unauthorized use.

Authenticity: Verifying users properly so they can use the program

Forward-secrecy: Changing of private keys and frequent encryption to make sure that even if one key is leaked, the leak of data will be as minimal as possible.

1. By using the shared secret key, the receiver can share the secret key that is used to create the message authentication code with the sender. Only the two parties can validate the message since they have the shared secret key, and if the message is somehow changed, the MAC key will change aswell.
2. By computing MAC keys from the encrypted messages, the attacker will only find the encrypted data if the MAC key is leaked, not the plain text. The attacker might not be able to decrypt it.

**Exercise 3:**

1. Since the developer lists every significant security related functionality they have implemented, I am going to assume that whatever was not mentioned might be vulnerability. He mentions the LoggedIn attribute being set to true, but says nothing about SameSite, HttpOnly or Secure attributes. The cookie headers might be configured incorrectly.

The website has no log in option for the users, only a distinct link which is connected to a user order. Without proper login and validation of the user, you cant be sure that it is actually the user himself navigating the site and sending the messages within their order.

Input field for users to send messages in directly to the database = potential SQL injection vulnerability. Prepared statements are not mentioned and might not be implemented. The storing of passwords should also be encrypted, I doubt it is. XSS is most likely possible as-well seeing as the input field is accessible to users and there is no mention of escaping html.

1. I would test SQL Injections by looking at how the input from user is sent to the database and by sending in various inputs and looking at the outcome. I would test the wrongfully user validation by trying to access some other users order and doing stuff on behalf of them. Tools such as OWASP ZAP and SonarQube would also be used in my testing process.
2. The most urgent one would be SQL Injections, seeing as the whole websites operation is run using the database. I would then look at securely storing passwords encrypted in the database, as well as implementing a log-in system for users with proper validation and headers of cookie. XSS would be less important than what mentioned above, but it is easy to fix so I would do that next. Properly securing the database would be highest priority.
3. Instead of sending user input directly to the database, I would escape all HTML and encode the input, as well as create prepared statements when creating SQL queries.

**Exercise 4:**

1. User can input their own value in the subforum variable, there is not escaping html in the code which means XSS might be possible.
2. Using XSS the attacker can display his own objects on the website, which can appear as perfectly normal buttons or links to the user, that can end up leaking document.cookie (users session cookie) to the attacker. The code could look something like this:

<button onclick=alert(document.cookie)>Click here! </button>

1. This can be prevented by checking every user input and escaping all the code. By doing this we make sure nothing is interpreted as code and only as regular string.

**Exercise 5:**

1. The CVE database is a collection of all kinds of different security threats that can occur. Every threat gets a CVE ID which tells us how critical the security issue is. Using a CVE database actively while protecting an application can be very useful. The developer can most likely think of the biggest security threats himself, but there might be some that he has missed, this is where a database which covers all the security threats can come in handy. It also makes the job much easier since you can easily arrange the different threats from least threatening to most threatening. Threats listed in the database needs to meet certain criteria, one cannot just shoot in a random threat, which means that there will be somewhat “quality” to the threats listed there.
2. OWASP ZAP and SonarQube are two very popular tools when trying to mitigate security issues.
3. The vulnerability is SQL Injection being possible by users in a specific file (db\_dump), and SQL statements can be ran with super user privileges. A user having super user privileges means that he can bypass all permission checks. It basically means admin powers.

This vulnerability can be very severe seeing as the user has admin powers and can pretty much do anything he wants. The CVSS v2.0 Severity and Metrics agrees with this. It looks like the fix checks if the old or new table has been deleted or equals null, if yes it will reference the respective table correctly.

**Exercise 6:**

1. As the administrators of a program we want to hand out the least amount of permissions. An attacker of a program wants the opposite to happen. The web-service can be tricked into giving an attacker privileges that he is not supposed to have in some cases. Requests can be forged and the program can end up giving the wrong users too much power. In another example there is a user that once had some privileges that later were removed. Special cache mechanisms can still give the user access to server-side data which he no longer is supposed to have access to.
2. This can be prevented using capability based access control since we can then access objects by having the designation to the object and the permission you would need to access that object working together.