# Answer Key with Justifications

## Network Security

| Q | Correct Answer | Justification |
|---|---|---|
| 1 | Smurf Attack | ICMP requests are sent to a broadcast address using the victim's spoofed IP, causing many replies to flood the victim. |
| 2 | Involvement of multiple compromised systems | DDoS attacks use multiple distributed sources (botnets), unlike single-source DoS attacks. |
| 3 | Coordinate and control compromised hosts | The C2 server issues commands to bots, coordinating attack timing and targets. |
| 4 | Consume bandwidth and network capacity | Volumetric attacks aim to saturate bandwidth rather than exploit vulnerabilities. |
| 5 | UDP | UDP is connectionless and easily spoofed, making it ideal for amplification attacks. |
| 6 | UDP lacks connection state and validation | No handshake or session tracking allows attackers to send massive traffic unchecked. |
| 7 | Ping of Death | Oversized ICMP packets cause buffer overflow and system crashes. |
| 8 | Sending packets to a broadcast address | Broadcast traffic amplifies replies from multiple hosts to one victim. |
| 9 | Redirecting traffic through the attacker | ARP poisoning enables MITM attacks by corrupting IP–MAC mappings. |
| 10 | Application Layer | HTTP floods and Slowloris attacks exploit application server behavior. |
| 11 | Appear similar to legitimate user traffic | Legit-looking requests make detection of application-layer attacks difficult. |
| 12 | Ingress filtering | Blocks incoming packets with spoofed source IP addresses. |
| 13 | Internal hosts launching spoofed attacks | Egress filtering prevents compromised internal hosts from attacking others. |
| 14 | Limiting traffic flow to manageable levels | Throttling controls traffic rate instead of completely blocking it. |
| 15 | Resource exhaustion | Dropping requests prevents CPU, memory, and session exhaustion. |
| 16 | Acting as a proxy for TCP handshakes | TCP Intercept completes the handshake on behalf of the server, blocking SYN floods. |

| | | |
|---|---|---|
| 17 | Load balancing | Traffic is distributed across multiple servers to maintain availability. |
| 18 | High-frequency request floods | Rate limiting restricts the number of allowed requests per source. |
| 19 | ARP Poisoning | Exploits trust in local address resolution rather than traffic volume. |
| 20 | Addresses attacks at multiple network layers | Layered defenses mitigate spoofing, flooding, and resource exhaustion together. |

## Access Control

| Q | Answer | Justification |
|---|---|---|
| 21 | b | Identification asks “Who are you?” (presenting an identity) while authentication verifies it. |
| 22 | c | Non-repudiation prevents a user from denying an action; auditing/accountability help, but non-repudiation is the mechanism. |
| 23 | b | Auditing is logging + periodic review of activities. |
| 24 | a | Accountability = holding a user responsible for actions tied to their identity. |
| 25 | b | Authorization = granting permissions after identity is established. |
| 26 | b | Storing templates and matching against the whole database is identification (1:N). |
| 27 | b | 1:N is the definition of identification (compare one probe to N enrolled). |
| 28 | b | Matching a fingerprint to the passport-linked record is verification (1:1). |
| 29 | b | False acceptance = an impostor (illegitimate user) is incorrectly accepted. |
| 30 | b | False rejection = a legitimate user is incorrectly denied. |
| 31 | None (correct ≈ 39.35%) — closest: c) 50% | p(per match)=0.0001% = 0.000001. For N=500,000: P(at least one) = 1−(1−p)^N ≈ 1−e^(−pN)=1−e^(−0.5)≈0.3935 ⇒ ≈39.35%. (None of the given options match; c)50% is the closest.) |
| 32 | b | Larger DB ⇒ more comparisons per identification → cumulative FAR increases (scales with N). |
| 33 | b | Identification (1:N) multiplies per-match FAR by many comparisons; FAR impact grows. |

| | | |
|---|---|---|
| 34 | d | A false acceptance grants unauthorized access → directly violates authorization (also impacts security broadly). |
| 35 | b | A criminal falsely accepted threatens national security; not merely performance. |
| 36 | b | False rejection mainly causes user inconvenience and delays (operational impact). |
| 37 | a | Matching against a list of criminals (searching across many identities) is identification. |
| 38 | b | False acceptance in blacklist check means a criminal is allowed to leave (false negative for the blacklist). |
| 39 | c | A whitelist grants privileged access → this is authorization (granting privileges). |
| 40 | b | False rejection on whitelist => authorized officials delayed or denied preferential treatment. |
| 41 | a | Office attendance tolerates false rejection (can clock manually) but cannot tolerate false acceptance as easily; airport control cannot. |
| 42 | c | Deleting biometric logs destroys auditing evidence (violates auditing requirement). |
| 43 | b | Biometrics + logs together strongly enforce accountability (who did what). |
| 44 | b | Biometrics are hard to share/forge, strengthening non-repudiation (you can tie actions to an individual). |
| 45 | c | Increasing *sensitivity* (raising matching threshold) decreases FAR but increases FRR — tradeoff. |
| 46 | d (correct ≈ 9.5%) | FAR per match = 0.001% = 0.00001. For 10,000 comps: pN = 0.00001×10000 = 0.1 ⇒ P(at least one)=1−e^(−0.1)≈0.09516 ⇒ ≈9.5%. Closest option: ~10% (d). |
| 47 | b | FAR = 0.0001 per attempt (decimal). Expected false accepts = attempts × FAR = 1000×0.0001 = 0.1. |
| 48 | b | Identification has higher overall false acceptance because you perform many comparisons (1:N); cumulative probability rises. |
| 49 | None (correct ≈ 86.5%) — closest: d) 2% | FAR per match = $1\times10^{-5}$. N=200,000 ⇒ pN = 2 ⇒ P(at least one)=1−e^(−2)≈0.8647 ⇒ ≈86.5%. (None of the options match; d)2% is not correct but closest numerically in list.) |
| 50 | b | Tightening the matching threshold (making acceptance stricter) reduces FAR but increases FRR. |

| | | |
|---|---|---|
| 51 | b | Lowering FRR (making system more permissive) raises FAR (more impostors accepted). |
| 52 | b | At high-security airport, false acceptance is more dangerous (allows threats through). |
| 53 | c | FRR = 2% ⇒ expected rejects = 0.02 × 5000 = 100. |
| 54 | c | MAC uses system-enforced security labels (mandatory policy), not owner discretion. |
| 55 | c | In DAC, the data owner (or owner process) controls access rights. |
| 56 | b | MAC is most resistant to insider abuse because the system enforces labels centrally. |
| 57 | b | In MAC, a "Secret" user cannot access "Top Secret" — denied by policy. |
| 58 | b | DAC allows users to grant/transfer their own permissions (subject to owner discretion). |
| 59 | b | MAC enforces centralized policy (labels handled by OS/kernel/policy). |
| 60 | c | Principle of Least Privilege = users get only the access they require. |
| 61 | c | Granting DB-admin rights for temporary debugging violates Least Privilege (excessive rights). |
| 62 | c | RBAC (Role Based Access Control) best supports enforcing least-privilege by assigning minimal roles. *(MAC also helps, but RBAC is typically used to operationalize least privilege.)* |
| 63 | b | A long-running service account with admin privileges violates least privilege and increases attack surface. |
| 64 | b | POLP primarily reduces insider threats and lateral movement after compromise. |
| 65 | b | Least Privilege limits damage after compromise by restricting what an account/process can do. |
| 66 | c | Switching to identification multiplies comparisons: risk (overall FAR) increases — mathematically P(at least one)=1−(1−p)^N (rises with N). |
| 67 | b | Blacklist (50,000) × FAR >> Whitelist (100) × FAR. Blacklist carries far higher aggregate FAR risk. |
| 68 | b | High threshold + verification (1:1) reduces FAR (strict matching) while keeping comparisons small — best balance for low FAR with reasonable usability (verification mode avoids 1:N explosion). |

# Web Security

| Q | Answer | Justification |
|---|---|---|
| 69 | d | Data redundancy is not a core goal; confidentiality, integrity, availability (CIA) are primary web security goals. |
| 70 | b | Protecting a web page from modification ensures integrity. |
| 71 | b | Preventing unauthorized access enforces authentication (verify user identity). |
| 72 | c | Website downtime due to heavy traffic compromises availability. |
| 73 | b | HTTP transmits data in plaintext; no encryption → confidentiality risk. |
| 74 | b | HTTPS secures communication via encryption and certificates. |
| 75 | c | HTTP traffic is secured using SSL/TLS (SSL legacy, TLS current). |
| 76 | c | Accessing HTTPS via HTTP leaves the connection vulnerable; encryption may not be applied. |
| 77 | b | GET appends parameters to the URL; POST sends data in the body. |
| 78 | b | GET parameters in URL can be logged, cached, and exposed → sensitive data leakage. |
| 79 | b | POST sends data in the request body, unlike GET which uses URL parameters. |
| 80 | b | POST is more suitable for login credentials to avoid URL exposure. |
| 81 | b | Host header specifies the server domain for HTTP/1.1 requests. |
| 82 | b | User-Agent identifies client software and version to the server. |
| 83 | b | Referer header reveals the page from which the request originated. |
| 84 | b | Cookies sent by client appear in the Cookie header; server sets via Set-Cookie. |
| 85 | b | SSL Strip downgrades HTTPS to HTTP to intercept traffic in plaintext. |
| 86 | c | SSL Strip requires a man-in-the-middle (MITM) to intercept and modify traffic. |
| 87 | b | Attack succeeds because users do not check the URL scheme (HTTP vs HTTPS). |
| 88 | b | TLS downgrade attacks force a connection to older, weaker encryption. |
| 89 | c | Legacy SSL versions (SSL 2.0 / SSL 3.0) are weak and targeted in downgrade attacks. |
| 90 | b | HSTS enforces HTTPS only, preventing downgrade attacks. |
| 91 | b | Mixed content occurs when an HTTPS page loads insecure HTTP resources. |

| | | |
|---|---|---|
| 92 | b | Active mixed content (scripts, iframes) can modify page behavior → most dangerous. |
| 93 | b | Mixed content compromises confidentiality and integrity, not availability directly. |
| 94 | b | HTTPS page submitting over HTTP is mixed content (secure page loading insecure form). |
| 95 | b | HSTS prevents SSL stripping by forcing browser to use HTTPS only. |
| 96 | b | Token-based authentication (Bearer token) uses the Authorization header. |
| 97 | c | Credentials visible in URL logs represent loss of confidentiality. |
| 98 | b | HTTPS alone cannot protect against mixed content vulnerabilities; insecure HTTP resources can be exploited. |

## SQL Injection

| Q | Answer | Justification |
|---|---|---|
| 99 | **b** | SQL Injection occurs due to improper validation and sanitization of user input. |
| 100 | **b** | Concatenating `$id` directly into the SQL query allows attackers to inject SQL code. |
| 101 | **a** | The payload bypasses login checks by making the authentication condition always TRUE. |
| 102 | **b** | URL query parameters are the most common injection point in GET-based SQL injection. |
| 103 | **b** | Google dorking helps attackers discover vulnerable web pages before exploitation. |
| 104 | **b** | `inurl:php?id=` identifies URLs likely using unsanitized SQL parameters. |
| 105 | **c** | `ORDER BY n` is used to determine the number of columns in the SQL query. |
| 106 | **b** | Error at column 6 means only 5 columns exist in the SELECT statement. |
| 107 | **a** | UNION queries require matching number of columns and compatible data types. |
| 108 | **b** | `version()` is a DB function commonly used to extract database version info. |
| 109 | **b** | UNION-based SQL injection exposes sensitive data, violating confidentiality. |
| 110 | **b** | Error-based SQL injection depends on visible database error messages. |
| 111 | **b** | Verbose exception handling leaks database details useful for attackers. |
| 112 | **c** | Blind SQL injection is used when responses are limited and no output/errors are shown. |
| 113 | **b** | Changes in page behavior indicate TRUE/FALSE evaluation in blind SQL injection. |
| 114 | **b** | Boolean-based injection observes logical differences in application responses. |
| 115 | **b** | Conditional content differences indicate boolean blind SQL injection. |

| | | |
|---|---|---|
| 116 | **c** | `SLEEP(5)` causes delayed responses to infer condition truth. |
| 117 | **c** | Time-based SQL injection is detected by noticeable delays in server response. |
| 118 | **b** | In second-order SQL injection, payload executes later when reused unsafely. |
| 119 | **b** | Stored malicious data executed later is the hallmark of second-order SQL injection. |
| 120 | **b** | Blacklists fail because attackers easily bypass fixed pattern-based filters. |
| 121 | **c** | Prepared statements separate SQL logic from user input, preventing injection. |
| 122 | **b** | Whitelisting allows only expected input formats, blocking malicious input. |
| 123 | **c** | SQL structure is parsed during the precompilation phase of prepared statements. |
| 124 | **b** | User input is treated strictly as data, not executable SQL code. |
| 125 | **b** | Dynamically concatenated table/column names can still allow SQL injection. |
| 126 | **c** | Blind SQL injection produces no direct output, making detection difficult. |
| 127 | **c** | Identical pages with delayed responses indicate time-based blind SQL injection. |
| 128 | **a** | Prepared statements preserve query structure, ensuring SQL integrity. |

## Cookies

| Q | Correct | Justification |
|---|---|---|
| 129 | **b** | HTTP does not retain state between requests, making it stateless. |
| 130 | **c** | Session IDs are the standard mechanism to maintain state across HTTP requests. |
| 131 | **c** | Session IDs link multiple requests to the same authenticated user session. |
| 132 | **b** | Cookies and session data enable personalization features like preferences. |
| 133 | **c** | Hidden fields pass state information between HTTP requests in forms. |
| 134 | **c** | Hidden fields can be viewed and modified by users, making them insecure alone. |
| 135 | **b** | `Set-Cookie` is the HTTP response header used to create cookies. |
| 136 | **c** | The Secure flag ensures cookies are sent only over HTTPS connections. |
| 137 | **b** | HttpOnly blocks JavaScript access, mitigating XSS-based cookie theft. |
| 138 | **a** | The Path attribute restricts cookie exposure to specific directories. |
| 139 | **b** | Cookies enable user tracking, which raises privacy and security concerns. |
| 140 | **c** | Third-party cookies track users across different websites and domains. |
| 141 | **c** | Browser fingerprinting identifies users without storing client-side data. |
| 142 | **b** | Screen resolution, fonts, and plugins are fingerprinting attributes. |
| 143 | **b** | Session hijacking occurs when attackers steal or guess session identifiers. |
| 144 | **b** | Cookie theft directly enables session hijacking attacks. |
| 145 | **b** | Session IDs in URLs can leak via logs, caches, and referrer headers. |
| 146 | **b** | HTTPS with Secure cookies prevents sniffing-based cookie theft. |
| 147 | **c** | Regenerating session IDs prevents session fixation attacks. |
| 148 | **b** | Secure + HttpOnly + HTTPS provides the strongest session cookie protection. |

# XSS and CSRF

| Q.No | Correct | Justification |
|---|---|---|
| 149 | **b** | SOP prevents scripts from accessing resources of other origins. |
| 150 | **b** | Origin = scheme (protocol) + host + port. |
| 151 | **c** | Different ports mean different origins under SOP. |
| 152 | **c** | Frame isolation blocks DOM access across frames of different origins. |
| 153 | **c** | Injected scripts run as if they belong to the site's origin. |
| 154 | **c** | XSS runs with the victim's permissions within that origin. |
| 155 | **c** | Stored XSS is saved on the server (DB, logs, comments). |
| 156 | **b** | Every visiting user is affected without extra interaction. |
| 157 | **b** | Server immediately reflects malicious input in the response. |
| 158 | **b** | Reflected XSS commonly appears in URLs or form submissions. |
| 159 | **b** | Injected scripts inherit origin, bypassing SOP intent. |
| 160 | **c** | CSP blocks execution of unauthorized scripts in browsers. |
| 161 | **b** | CSP restricts inline scripts and untrusted sources. |
| 162 | **b** | `script-src` controls allowed JavaScript sources. |
| 163 | **b** | Input filters are error-prone and easily bypassed. |
| 164 | **b** | Browsers automatically attach cookies to requests. |
| 165 | **a** | SOP restricts reading responses, not sending requests. |
| 166 | **b** | CSRF often tricks victim into submitting authenticated requests. |
| 167 | **b** | Server checks where the request originated from. |
| 168 | **b** | Browsers may suppress Referer for privacy/security reasons. |
| 169 | **b** | Secret per-request tokens prevent forged requests. |
| 170 | **c** | Tokens must be unpredictable to prevent guessing. |
| 171 | **b** | Server compares cookie token with request token. |
| 172 | **a** | No server-side token storage is required. |
| 173 | **b** | Lax allows cookies on top-level navigations (GET). |
| 174 | **b** | Strict prevents cookies from being sent cross-site entirely. |
| 175 | **c** | Strict maximizes security but reduces usability. |
| 176 | **c** | Token + SameSite provides layered CSRF defense. |