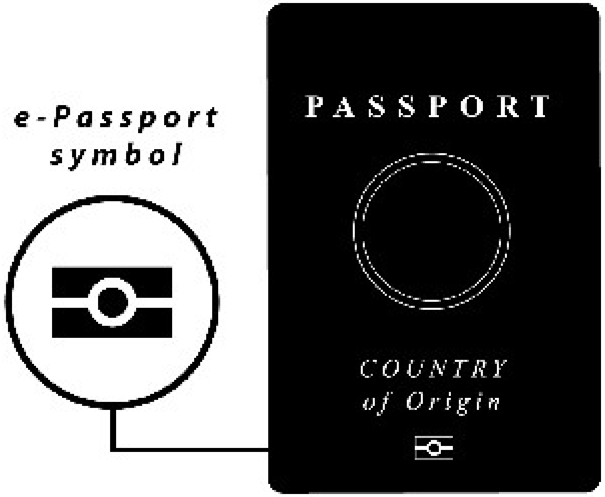
Authentication and Session Management



Authentication Basics

- There are 3 methods of identifying an individual.
  - Something you have – e.g. token, certificate, cell
  - Something you are – e.g. biometrics
  - Something you know – e.g. password
- For highly sensitive applications multifactor authentication can be used
- Financial services applications are moving towards “stronger authentication”
- Google is a good example of a free consumer SaaS service that offers multi-factor authentication

Session Identifiers

- Once a user has proven their identity, session management functionality is employed
- Each request sent to the server contains an identifier that the server uses to associate requests authenticated users
- The Session identifier is all that is need to prove authentication for the rest of the session
- Keeping Session IDs secure is critical
- Session ID's are typically passed in one of three places:
  - URL query string
  - Hidden Form Field
  - Cookie HTTP Header
- In general, this is transparent to the user and is handled by the web server

Broken Session Management

- The client can never be trusted
- The client cannot be relied upon for providing or ensuring security
- The HTTP protocol does not have an innate method of state-management
- Anything deployed on the client-side is susceptible to offline attacks
- Data stored on the client must be protected from unauthorized viewing or tampering
- Avoid passing session ID's in the URL Query string (session rewriting)

Authentication Dangers

- Passwords & PIN’s
  - Subject to brute-force attack
  - Favorite words often used , weak passwords
  - Users share with others
  - Plaintext or poor password storage
- Certificates
  - Attackers obtain certificate files
  - Not all CA’s are trustworthy
- Biometrics
  - Subject to Replay attacks
  - False/Positive and False/Negative errors

More Authentication Dangers

- Session Management Weaknesses
  - Session Fixation
  - Weak or Predictable Session
  - Session Hijacking via XSS
  - Session Hijacking via network sniffing
- Username Harvesting
  - Registration page makes this easy
- Weak "Forgot Password” feature
  - Reset links sent over email
- Weak "Change Password” feature
  - Does not require existing password
  - Access control weakness allows reset of other users password

Login Functionality Attacks

- **Username enumeration** which allows an attacker to enumerate valid usernames for use with further attacks
- **Password guessing** which is most successful when users are allowed to choose weak passwords
- **Brute-Force Attacks** which succeeds when there is no account lockout or monitoring of login attempts
- **Credential Theft** which succeeds when there is no or poor encryption protecting credentials stored or in transit

Attacks Against Session Identifiers

- If session identifiers are issued in a predictable fashion, an attacker can use a recently issued Session ID to guess other valid values
- If the possible range of values used for Session ID’s is small, an attacker can brute force valid values
- Session ID’s are also susceptible to disclosure via network sniffing attacks
- Once obtained, a session ID typically allows impersonation of the user
  - Susceptible to replay
  - No need to steal user credentials

Credential Defenses

- Various aspects the application should require the user to provide proof of identity
  - Login
  - Password Reset
  - Shipping to a new address
  - Changing email address or other user profile items
  - Significant or anomalous transactions
  - Helps minimize CSRF and session hijacking attacks
- Implement server-side enforcement of password syntax and strength (i.e. length, character requirements, etc)
  - Helps minimize login password guessing

Additional Authentication Best Practices

- Where possible restrict administrator access to machines located on the local area network (i.e. it’s best to avoid remote administrator access from public facing access points)
- Log all failed access authorization requests to a secure location for review by administrators
- Perform reviews of failed login attempts on a periodic basis
- Utilise the strengths and functionality provided by the SSO solution you chose, e.g. Netegrity

Login and Session Defenses

- Send all credentials and session id's over well configured HTTPS/SSL/TLS
  - Helps avoid session hijacking via network snifing
- Develop generic failed login messages that do not indicate whether the user-id or password was incorrect
  - Minimize username harvesting attack
- Enforce account lockout after a pre-determined number of failed login attempts
  - Stops brute force threat
- Account lockout should trigger a notification sent to application administrators and should require manual reset (via helpdesk)

More Session Defenses

- Ensure that Session ID values are not predictable and are generated from a large range of possible values
  - 20+ bytes, cryptographically random
  - Stored in HTTP Cookies
  - Cookies: Secure, HTTP Only, limited path
  - Helps avoid session id guessing or hijacking threat
- Generate new session ID at login time
  - To avoid *session fixation* threat
- Session Timeout (sessions must “expire”)
  - Idle Timeout due to inactivity
  - Absolute Timeout
  - Logout Functionality
  - Will help minimize session hijacking threat

Logout / Session Defenses

- Give users the option to log out of the application and make the option available from every application page
- When clicked, the logout option should prevent the user from requesting subsequent pages without re-authenticating to the application
- The user’s session should be terminated using a method such as session.abandon(), session.invalidate() during logout
- Users should be educated on the importance of logging out, but the application should assume that the user will forget
- JavaScript can be used to force logout during window close event

Password Defenses

- Disable Browser Autocomplete
  - <form AUTOCOMPLETE="off”>
  - <input AUTOCOMPLETE="off”>
- Only send passwords over HTTPS POST
- Do not display passwords in browser
  - input type=password
  - Do not display passwords in HTML document
- Store password on server via one-way encryption
  - Hash password
  - Use Salt
  - Iterate Hash many times

Password Storage Code Sample

```
public String hash(String plaintext, String salt, int iterations)
     throws EncryptionException {
byte[] bytes = null;
try {
  MessageDigest digest = MessageDigest.getInstance(hashAlgorithm);
  digest.reset();
  digest.update(ESAPI.securityConfiguration().getMasterSalt());
  digest.update(salt.getBytes(encoding));
  digest.update(plaintext.getBytes(encoding));

  // rehash a number of times to help strengthen weak passwords
  bytes = digest.digest();
  for (int i = 0; i < iterations; i++) {
     digest.reset();  bytes = digest.digest(bytes);
   }
  String encoded = ESAPI.encoder().encodeForBase64(bytes,false);
  return encoded;
} catch (Exception ex) {
       throw new EncryptionException("Internal error", "Error");
}}
```

Forgot Password Secure Design

- Require identity questions
  - Last name, account number, email, DOB
  - Enforce lockout policy
- Ask one or more good security questions
  - http://www.goodsecurityquestions.com/
- Send the user a randomly generated token via out-of-band communication
  - email, SMS or token
- Verify code in same web session
  - Enforce lockout policy
- Change password
  - Enforce password policy

Encryption in Transit (TLS)

- Authentication credentials and session identifiers must me be encrypted in transit via HTTPS/SSL
  - Starting when the login form is rendered
  - Until logout is complete
  - All other sensitive data should be protected via HTTPS!
- https://www.ssllabs.com free online assessment of public facing server HTTPS configuration
- https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet for HTTPS best practices

Insecure Use of HTTP Cookies

- Cookies provide a means of storing data that will be sent by the user with every HTTP request
- Persistent cookies are stored on the users hard drive, potentially exposing them to unauthorised access
- While cookies can be safe when used responsibly, some applications store information in cookies that is easily modified
- Interception or modification of cookies that are not cryptographically secure could allow an attacker to:
  - Gain access to unauthorized information
  - Perform an activity on behalf of other users
  - Not as widespread as used to be

Cookie Options

The Set-Cookie header uses the following syntax:

Set-Cookie: *NAME*=*VALUE*; expires=*DATE*; path=*PATH*; domain=*DOMAIN_NAME*; secure

- Name
  - The name of the cookie parameter
- Value
  - The parameter value
- Expires
  - The date on which to discard the cookie (if absent, the cookie not persistent and is discarded when the browser is closed.

Cookie Security Defenses

- Path
  - The path under which all requests should receive the cookie. “/” would indicate all paths on the server
- Domain
  - The domain for which servers should receive the cookie (tail match). For example, my.com would match all hosts within that domain (www.my.com, test.my.com, demo.my.com, etc.)
- Secure
  - Indicates that the cookie should only be sent over HTTPS connections
- HTTPOnly
  - Helps ensure Javascript can not manipulate the cookie. Good defense against XSS.

Cookie Security Defenses

- Avoid storing sensitive data in cookies
- Avoid using persistent cookies
- Always set the “secure” cookie flag for HTTPS cookies to prevent transmission of cookie values over unsecured channels
- Any sensitive cookie data should be encrypted if not intended to be viewed/tampered by the user. Persistent cookie data not intended to be viewed by others should always be encrypted.
- Cookie values susceptible to tampering should be protected with an HMAC appended to the cookie, or a server-side hash of the cookie contents (session variable)

Session Management Code Review Challenge

Challenge!

Examine the following Pseudo code and identify any issues with this session management mechanism.

Pseudo Code: Session Creation, Authorization, Session Validation

ROW	CODE	FIX? Y / N
1	BROWSER requests access to “Account Summary” from WEBSERVER	
2	WEBSERVER checks whether the session is authenticated	
3	IF session is authenticated:	
4	Send “Account Summary” page to BROWSER	
5	RETURN	
6	IF session is NOT authenticated:	
7	WEBSERVER grabs USERNAME posted by BROWSER	
8	WEBSERVER asks DATABASE (“Select * from AuthTable where Username = ‘%s’”, USERNAME);	
9	IF DATABASE returns no users:	
10	WEBSERVER sends error message to BROWSER (“Invalid User Name %s”, USERNAME);	
11	RETURN	
12	ELSE	
13	WEBSERVER grabs PASSWORD posted by BROWSER	
14	For each user returned by DATABASE:	
15	IF user’s password equals PASSWORD:	
16	Authenticate session	
17	Generate Session ID:	
18	Increment previous Session ID by 1	
19	Store Session ID	
20	Add Session ID to user’s cookie	
21	IF no users have a password equal to PASSWORD:	
22	WEBSERVER sends error message to Browser (“Invalid password %s for username %s”, PASSWORD, USERNAME);	

Solution

```
BROWSER requests access to "Account Summary" from WEBSERVER
WEBSERVER checks whether the session is authenticated
  IF session is authenticated:
    Send "Account Summary" page to BROWSER
    RETURN
  IF session is NOT authenticated:
    WEBSERVER grabs USERNAME and PASSWORD posted by BROWSER
    WEBSERVER asks DATABASE ("Select * from AuthTable where Username = '%s' and
      Password = '%s'", USERNAME, PASSWORD);
    IF DATABASE returns no users or more than one user:
      WEBSERVER sends error message to BROWSER ("Invalid User Name or Password");
      RETURN
    ELSE (DATABASE has returned exactly one user)
          Authenticate session
          Generate Session ID:
              WEBSERVER generates secure Session ID
      Store Session ID
      Add Session ID to user's cookie
```