ENPM 809W Introduction to Secure Software Engineering

Gananand Kini

Lecture 10

Session Management related security bugs - Defenses

Outline - Secure Session Management

- **Secure Session Management**
  - Countering Session Fixation attacks
- **Countering Cross-Site Request Forgeries**
- **Hide sensitive information**
- **Security configurations**

Secure Session Management

Session

- **Does your software system really need state?**
  - Only use sessions if they are required!
  - No need for sessions for a read-only static blog for example.
- **If the software system requires session and to keep state:**
  - Only store essential variables in the state.
  - Try to protect session identifiers and any sensitive user data
  - Enforce lifetimes on the session (make them expire after a certain period)!
    - General rule: 30 minute timeout after inactivity, 12 hour hard timeout
  - Session management settings are typically given as part of the application server or platform configuration. As software engineers, need to understand how these settings affect the running of the software system and verify whether the administrator has the settings correctly configured.

Session Identifiers and Lifetimes

- Use unique non-predictable session identifiers.
- Set an expiration (lifetime) to the session itself (along with all the values in the session state)
- Regenerate session identifiers by either overwriting existing or creating a new identifier for authentication/authorization purposes.

Session Fixation Example

```
public int authenticate (HttpSession session)
{
   string username = GetInput("Enter Username");
   string password = GetInput("Enter Password");

   // Check maximum logins attempts
   if (session.getValue("loginAttempts") > MAX_LOGIN_ATTEMPTS)
   {
      lockAccount(username);
      return(FAILURE);
   }

   if (ValidUser(username, password) == SUCCESS)
   {
      session.putValue("login", TRUE);
      return(SUCCESS);
   }
   else return(FAILURE);
}
```

In order to exploit the code above, an attacker could first create a session (by visiting the login page of the application) from a public terminal, record the session identifier assigned by the application, and then leave the login page open. Next, a victim sits down at the same public terminal, notices the browser open to the login page of the site, and enters credentials to authenticate against the application. The code responsible for authenticating the victim continues to use the pre-existing session identifier, now the attacker simply uses the session identifier recorded earlier to access the victim's active session, providing nearly unrestricted access to the victim's account for the lifetime of the session.

Session Fixation Example: Potential Fix

```
public int authenticate (HttpSession session)
{
   string username = GetInput("Enter Username");
   string password = GetInput("Enter Password");

   // Check maximum logins attempts
   if (session.getValue("loginAttempts") > MAX_LOGIN_ATTEMPTS)
   {
      lockAccount(username);
      return(FAILURE);
   }

   if (ValidUser(username, password) == SUCCESS)
   {
      // Kill the current session so it can no longer be used
      session.invalidate();

      // Create an entirely new session for the logged in user
      HttpSession newSession = request.getSession(true);

      newSession.putValue("login", TRUE);
      return(SUCCESS);
   }
   else return(FAILURE);
}
```

Session Fixation in ASP .NET: Potential fix

```
protected void Page_Load(object sender, EventArgs e)
{
    //NOTE: Keep this Session and Auth Cookie check
    //condition in your Master Page Page_Load event
    if (Session["LoggedIn"] != null &&
Session["AuthToken"] != null
           && Request.Cookies["AuthToken"] != null)
    {
        if (!Session["AuthToken"].ToString().Equals(
                   Request.Cookies["AuthToken"].Value))
        {
            // redirect to the login page in real
application
            lblMessage.Text = "You are not logged in.";
        }
        else
        {
            lblMessage.Text = "Congratulations !, you are
logged in.";
            lblMessage.ForeColor =
System.Drawing.Color.Green;
            btnLogout.Visible = true;
        }
    }
    else
    {
        lblMessage.Text = "You are not logged in.";
        lblMessage.ForeColor = System.Drawing.Color.Red;
    }
}
```

```
protected void LoginMe(object sender, EventArgs e)
{
    // Check for Username and password (hard coded
for this example)
    if (txtU.Text.Trim().Equals("u") &&
              txtP.Text.Trim().Equals("p"))
    {
        Session["LoggedIn"] = txtU.Text.Trim();
        // createa a new GUID and save into the
session
        string guid = Guid.NewGuid().ToString();
        Session["AuthToken"] = guid;
        // now create a new cookie with this guid
value
        Response.Cookies.Add(new
HttpCookie("AuthToken", guid));

    }
    else
    {
        lblMessage.Text = "Wrong username or
password";
    }
}
```

```
protected void LogoutMe(object sender, EventArgs e)
{
    Session.Clear();
    Session.Abandon();
    Session.RemoveAll();

    if (Request.Cookies["ASP.NET_SessionId"] !=
null)
    {
        Response.Cookies["ASP.NET_SessionId"].Value
= string.Empty;

Response.Cookies["ASP.NET_SessionId"].Expires =
DateTime.Now.AddMonths(-20);
    }

    if (Request.Cookies["AuthToken"] != null)
    {
        Response.Cookies["AuthToken"].Value =
string.Empty;
        Response.Cookies["AuthToken"].Expires =
DateTime.Now.AddMonths(-20);
    }
}
```

Sources:

1. ITFunda. Session Fixation Vulnerability in ASP.NET. https://www.codeproject.com/articles/210993/session-fixation-vulnerability-in-asp-net.

Preventing session fixation

- Try to obtain a new session identifier on Login as well.

- A bit involved to try and do it in ASP .NET Core since it uses middlewares for session management (will talk about later).

Preventing Cookie Fixation

- **HTTPSessionState.Clear() method:**
  - https://learn.microsoft.com/en-us/dotnet/api/system.web.sessionstate.httpsessionstate.clear?view=netframework-4.8#system-web-sessionstate-httpsessionstate-clear
  - Does this really clear the session token and identifier from the cookie?
- **From https://www.c-sharpcorner.com/article/asp-net-core-working-with-cookie/:**

```
foreach (var cookie in Request.Cookies.Keys) { Response.Cookies.Delete(cookie); }
```

Countering Cross-Site Request Forgeries (CSRF/XSRF)

Countering Cross-Site Request Forgeries (CSRF/XSRF)

- **An issue of authentication and authorization**
- **Require re-authentication**
- **Use “SameSite” cookie attribute set to ‘Lax’ (Default) or ‘Strict’, not ‘None’.**
- **Use cookie attribute “Secure” to ensure cookies are only sent over HTTPS.**
- **Prevent leaks of session identifiers to scripts by setting ‘HTTPOnly’ flag to True.**
- **Set Domain if possible to only scope cookies to a single domain.**
- **Check HTTP Referer or Origin Header but don’t rely on it.**
- **Use CSRF countermeasures in login to prevent login forgery.**
- **Defense-in-depth measures:**
  - Logoff after X minutes of inactivity
  - Do not allow GET link or URL to have side-effects – only POST (button)
- **Secure Defaults**
  - Browsers now have countermeasures built-in

CSRF/XSRF: A success story

- **At one time CSRF/XSRF was one of the most common serious vulnerabilities**
- **Drastically reduced likelihood today, because countermeasures now enabled by default**
  - Frameworks typically embed countermeasure.
  - Cookie “SameSite” setting provided simple countermeasure.
  - Chrome & Firefox switching to SameSite=Lax by default, preventing the problem entirely for them.
- **Illustrates the value of defaults**
  - Where possible, build countermeasures into your tools/standards/system so the problem won’t occur

ASP .NET Core – Preventing CSRF

- The new ASP .NET Core with Razor pages has XSRF/CSRF protection built in.
- See https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-5.0.
- Uses anti-forgery tokens generated by the server and makes the round trips between server and client. It's unique and un-guessable :D
- The token is added as part of a "hidden" field.

- What happens if the token is leaked (no SSL for example)?

Web applications – hardening headers to make attacks harder

- **Content Security Policy (CSP) – already noted**
- **HTTP Strict Transport Security (HSTS)**
  - “Only use HTTPS from now on for this site”
- **X-Content-Type-Options (as "nosniff")**
  - Don’t guess MIME type (& confusion it can bring)
- **X-Frame-Options**
  - Clickjacking protection, limits how frames may be used
- **X-XSS-Protection**
  - Force enabling filter to detect likely (reflected) XSS by monitoring requests / responses
  - On by default in most browsers
- **Quick scan tool: https://securityheaders.io/**

See: https://www.owasp.org/index.php/List_of_useful_HTTP_headers

Double Submit Cookie technique

- **Not perfect, but better than current STP implementation in ASP .NET Core:**
  - https://learn.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-3.1#aspnet-core-antiforgery-configuration
- **Assumes strict transport (HSTS) and strict cookie policies talked about earlier in preventing CSRF.**
- **On first request to webserver from client, you generate a random nonce and produce an HMAC using a strong key based hash (like SHA-512).**
- **Send the HMAC in the response as part of the first cookie being sent.**
- **Client then has to send that same HMAC value with every request embedded in headers as well as the cookie.**
- **Server on pages that need to prevent CSRF checks value in cookie is same as in header.**
- **David Johansson shows attacks against this when the above assumptions of strict policies are not followed: https://owasp.org/www-pdf-archive/David_Johansson-Double_Defeat_of_Double-Submit_Cookie.pdf**
- **You could add a third check: Check HMAC value in cookie, in header and the one generated server side to get around the above attack.**
- **Definitely complicated to implement in ASP .NET Core but not impossible.**

ASP .NET Core Resources for Session/State Management

- **Session and state management in ASP .NET Core:**
  - https://learn.microsoft.com/en-us/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1

- **Setting and retrieving Cookies in ASP .NET Core:**
  - https://learn.microsoft.com/en-us/aspnet/web-api/overview/advanced/http-cookies#example-set-and-retrieve-cookies-in-a-message-handler

- **How to make your own middleware (for example to implement your own custom cookie/session handling)**
  - https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1
  - https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1#middleware-order-1

Also JWT tokens for sessions are NOT the answer…

- JSON Web Tokens are used by OAuth to provide authentication and authorization.
- Why not use it for managing sessions too?
- Lots of people suggesting uses for Session management as well. See: https://supertokens.io/blog/are-you-using-jwts-for-user-sessions-in-the-correct-way
- Managing them is a pain, unlike OAuth where a dedicated server exists to manage them.
- You are hopefully using OAuth to manage transactions, not full time use of APIs or microservices.
- Revoking and managing session tokens becomes tricky.
- Sessions are stateful, whereas JWT use case is only for stateless one time type of transactions.
- For why not to use JWT for session management see http://cryto.net/%7Ejoepie91/blog/2016/06/19/stop-using-jwt-for-sessions-part-2-why-your-solution-doesnt-work/.

Apply Defense-in-Depth principles…

- Use but don’t trust client provided information about sessions such as session identifiers, client identifiers etc.
- Re-authenticate and re-authorize for sensitive operations.
- Manage session lifetimes (from session creation, verification to destruction) in accordance with threat profiles. Use inactivity timeouts as well as hard timeouts.
- Watch out for information leak possibilities such as session identifier or tokens leaking as part of a GET request. Restrict sending session information to easily accessible requests. Do not send sensitive information as part of URLs.
- Appropriately configure software system parts to be accessible only when authenticated or as part of a session. Use an all or none approach. Use a session for all requests or none of them. Using sessions only for some parts of the software system can lead to misconfiguration issues and can be dangerous and would need to be managed carefully.

Apply Defense-in-Depth principles…

- **Generate cryptographic material fresh on session creation. Secret keys, and Initialization Vectors (IVs) should be recreated upon session creation.**

Next time …

- Error Handling and Logging bugs - Attacks