We use sessions, because web is running on HTTP, which is a stateless protocol. However, in many web applications, we need to keep some state information, some context. This is the purpose of sessions. When a user opens a browser and navigates to some website, the web server of that website generates a session ID for this user. The session ID is sent with every request and it is a key for any session data (session data = state/context). These data are stored on the server.

The session ID can be passed to the requests in two ways:

- via a URL parameter,
- or using a cookie.

It is recommended to use cookies for passing the session ID. You can disable the cookieless sessions by changing the cookieless attribute of the form element on the **<system.web>** section of the web.config file:

```
<authentication mode="Forms">
        <forms cookieless="UseCookies" />
```

The session ends (in a typical case) after the user closes the browser or after the user is inactive for a specified amount of time.

There are basically three types of session attacks:

- Session stealing
- Session prediction
- Session fixation

Session stealing

A session can be stolen by stealing a session ID. When an attacker steals a user's session ID, he can get access to all of the session data. Because all these session IDs can be read by JavaScript, the most popular method for this type of attack is XSS. An attacker can send a crafted link to a victim with a malicious JavaScript. When the victim clicks on the link, the JavaScript runs and sends the cookie value of the current session to the attacker.

Session prediction

Can an attacker simply guess some random session ID? Most implementations of session IDs are long strings and guessing a correct session ID in linear time is impossible. But there are also bad implementations when an attacker can generate session IDs from known values. This technique is called session prediction. For example, a session ID can be a user name encoded in base64. Fortunately, in ASP.NET, session ID is a 120bit random number represented by a 20-character string. So, it is relatively safe.

Session fixation

In this case, the attacker lets the server generate a session ID. Then the attacker sets a user's session ID to the generated ID. This is quite easy when the session ID is given in a URL parameter. After that, the user and the attacker share the same session. For example, when the user gets authenticated, the attacker is also authenticated as the user.

The goal of all kinds of session attacks is the same – to get the user's session data or achieve identity forgery. This page focuses mainly on **session fixation**, as XSS is explained on a different page and we cannot influence how session IDs are generated. But generally, the implications of all three session attacks and the protective measures against them are similar.

Example of a session fixation

Let's have a simple .aspx page which saves a value to a session and also shows it:

```
<asp:Literal runat="server" ID="ltlSession"></asp:Literal>
<asp:TextBox runat="server" ID="txtValue"></asp:TextBox>
<asp:Button runat="server" ID="btnSend" Text="Save" />
```

In the code behind, we handle the OnClick() event of the button:

```
Session["MyPrivateData"] = txtValue.Text;
```

On page load, we display the value via the literal:

```
ltlSession.Text = "My private Data:" + CMS.Helpers.SessionHelper.GetValue
("MyPrivateData") + "<br/>";
```

How the attack is executed:

1. The attacker forges a link to this page with a session ID and sends the link to the user.
2. The user, unaware of the forgery, clicks the link.
3. The user sees the page normally and does not register anything unusual.
4. When the user saves some private data now, the attacker will be able to access it.

What can session fixation attacks do

The main goal of the attacker is to read and manipulate session data or forge their identity. In both cases, the risks depend on the implementation of individual applications. If the application stores sensitive data to sessions (for example, user passwords in plain text) and allows this data to be displayed or changed, the damage cane be severe.

Finding session fixation vulnerabilities

In Kentico, session fixation is possible, depending on the application settings. You have to ensure that you do not store any sensitive information in sessions. The best way is to determine which variables are stored in sessions. Then, check how you can manipulate with them (read/change) and think about the risks – what damage can the attacker do by manipulating them. You can find these risks just from the user perspective by inspecting application reactions, parameters and so on.

However, we still recommend code inspection. You can simply find all manipulations with session data by searching for **Session Helper** and the **Session[]** array.

Avoiding session fixation

First of all, there is no native support for protecting against session fixation in ASP.NET. The best practice for protecting your application against session fixation is to regenerate the session ID after a user logs on. You can achieve this by changing the session ID to an empty string and letting ASP.NET generate a new one. However, this action causes loss of session data.

In Kentico, you can include the **CMSRenewSessionAuthChange** key in the **appSettings** section of your web.config file, which enforces a change of the session ID when a user logs in or out. If you enable this setting, users will not be able to preserve their session data after logging in or out:

```
<add key="CMSRenewSessionAuthChange" value="true" />
```

It is still possible for attackers to manipulate non-user session data. You need to ensure that you **do not save sensitive and critical information in sessions**.