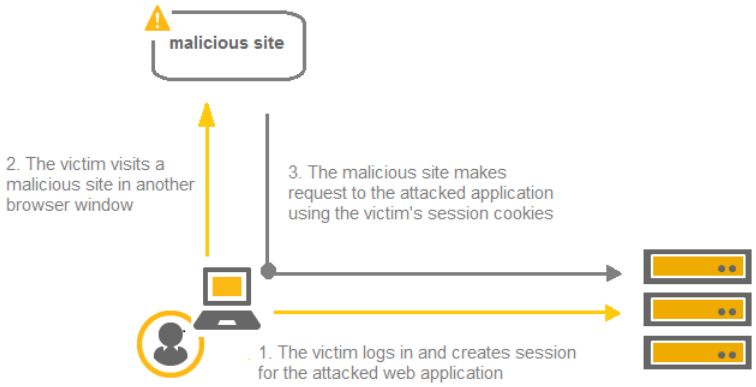
CSRF- Cross Site Request Forgery

Bedrohung

Mittels Cross Site Request Forgery kann der Angreifer den User dazu bringen, unerwünschte Requests an andere Webapplikationen zu schicken. Dabei wird auf der Webseite des Angreifers JavaScript-Code ausgeführt, welcher unbemerkt einen Request an die Webapplikation schickt. Beim Request wird das gültige Session-Cookie des Users angehängt. Die Webapplikation geht davon aus, dass der User diesen Request selbst abgeschickt hat, da das Session-Cookie vorhanden ist. Ist Cross-Site-Resource-Sharing nicht explizit erlaubt, kann der Angreifer kann nur Requests ausführen, aber ihre Antwort nicht lesen und zielt somit auf Statusänderungen ab.



Schutzmassnahmen

Es gibt viele verschiedene Möglichkeiten, um sich vor einem CSRF-Angriff zu schützen, doch nützen diese alle nichts, solange man nicht ebenfalls Massnahmen gegen Cross Site Scripting führt. **Alle CSRF-Schutzmassnahmen können von XSS umgangen werden.** Im folgenden Teil werden einige dieser Schutzmassnahmen aufgeführt. Ausserdem ist es Applikationsabhängig, welche Schutzmassnahmen zu verwenden sind.

Framework

Viele grosse Frameworks bieten bereits CSRF-Schutz. Als erstes sollte immer auf die Implementierung des Frameworks zurückgegriffen werden, bevor man diese selbst vornimmt oder ergänzt.

Synchronizer Pattern

Dieses Pattern ist die bekannteste Methode, um sich zu schützen. Der Server generiert ein CSRF-Token, dass vom Client bei jedem Request als Custom-Header oder in einem Hidden-Form-Field mitgeschickt werden muss. Das Token wird in der Datenbank per User abgelegt, diese Methode ist somit nicht stateless. Der Sever hat nun die Möglichkeit das Token aus dem Request mit dem Token aus der Datenbank zu überprüfen. Das Token sollte in einem Header und nicht mittels Cookies übertragen werden, denn sonst kann auch dieses automatisch vom Angreifer mitgeschickt werden. Bei GET-Requests sollte man auf die Übertragung des Tokens verzichten, da dies bei GET-Request geleakt werden könnte, was aber bedingt, dass keine Statusänderungen mit GET-Requests ausgeführt werden können.

Stateless Protection
Ist es nicht möglich oder nicht gewünscht, dass CSRF-Token in einer Datenbank zu speichern, so gibt es noch andere Varianten zum CSRF-Token.

Double Submit Cookie
Mit dieser Methode wird zusätzlich zum Session-Cookie ein weiteres Cookie mit einem Token generiert. Dieses Token wird bei jedem Request zusammen mit dem Session-Cookie mitgeschickt. Das Token wird zusätzlich in einem Custom-Header oder in einem Hidden-Form-Field platziert. Der Server kann nun den Request validieren, indem er prüft, ob das Token im Header oder im Body das gleiche wie im Cookie ist. Dies funktioniert, da der Angreifer mit JavaScript keine eigenen Request-Header setzen kann. Diese Methode hat aber noch einen Mangel. Hat die Domain noch Subdomains, sollte das Cookie zusätzlich verschlüsselt werden, da dieses sonst bei einem Angriff auf eine Subdomain überschrieben werden könnte.

CSRF-Token mit JWT
Json Web Tokens haben den Vorteil, dass sie für stateless Sessions verwendet werden können. Anstatt das CSRF-Token in der Datenbank zu hinterlegen, kann das Token im verschlüsselten JWT-Token platziert werden. Zusätzlich wird es im Localstorage des Clients gespeichert, welcher das CSRF- und JWT-Token beim Request an den Server mitsendet. Das CSRF-Token schickt man am besten über den Custom X-CSRF-Token Header. Der Server kann nun einfach das JWT entschlüsseln und das Token darin mit dem Token im Header vergleichen.

User Interaction based
Diese Methode bietet den höchsten Schutz aber ein schlechteres Nutzungserlebnis. Vom User wird eine erneute Bestätigung mittels CAPTCHA, One-Time Token oder erneuten Anmeldung angefordert, bevor eine Aktion durchgeführt wird.

Demo

Angriff
Der Angreifer startet von http://localhost:3000 seinen Angriff auf die Webapplikation http://localhost:8020/csrf/delete. Das Opfer hat bereits ein gültiges Cookie, mit welchem es auf der Webapplikation Requests ausführen kann.

Name	Value	Domain	Path
token	asdfou9x7yc9vyxkcovkjh	localh...	/

Das Opfer öffnet die Webseite des Angreifers, welche folgenden JavaScript-Code ausgeführt.

```html
<!DOCTYPE html>
<html>
    <head>
        <meta
        http-equiv="Content-Type"
         content="text/html; charset=UTF-8">
    </head>
<body onload="post()">
</body>
</html>
```

```html
<script>
    function post() {
        var x = new XMLHttpRequest();
        x.open("POST",
        "http://localhost:8020/csrf/delete",
         true);
        x.withCredentials = true;
        x.setRequestHeader("Content-Type",
         "application/json");
        x.setRequestHeader("X-CSRF-TOKEN",
         localStorage.getItem("X-CSRF-TOKEN"));
        x.send(JSON.stringify({ "id": "2" } ));
    }
</script>
```

Beim Request wird automatisch durch «withCredentials» das Cookie mitgeschickt. Die Webapplikation verwendet folgenden Code zur Überprüfung des Tokens.

```php
public function delete($requestProvider)
{
    $origin = $requestProvider->getParam("HTTP_ORIGIN");
    header("Access-Control-Allow-Origin: $origin");
    header("Access-Control-Allow-Credentials: true");
    header('Access-Control-Allow-Headers: Content-Type');

    try {
        $id = $requestProvider->getBodyParam("id");
        $token = $requestProvider->getCookieParam("token");
        if (Authorization::decodeAccessToken("Bearer " . $token)) {
            return $this->response(static::class, "Delete object with id $id.");
        }
        $this->error("Token $token is invalid.", 401);
    } catch (Exception $e) {
        throw new Exception($e->getMessage(), 400);
    }
}
```

Da das Cookie mit dem gültigen Token im Request vorhanden ist, wurde das Objekt mit Id 2 gelöscht.

Schutz

Da wir in dieser Applikation JWT verwenden, bietet es sich an, das CSRF-Token im JWT zu hinterlegen. In der Webapplikation kann nun überprüft werden, ob das CSRF-Token im X-CSRF-TOKEN Custom-Header das gleiche wie im entschlüsselten JWT-Token ist.

```php
public function delete($requestProvider)
{
    $origin = $requestProvider->getParam("HTTP_ORIGIN");
    header("Access-Control-Allow-Origin: $origin");
    header("Access-Control-Allow-Credentials: true");
    header('Access-Control-Allow-Headers: Content-Type');

    try {
        $id = $requestProvider->getBodyParam("id");
        $token = Authorization::decodeAccessToken("Bearer " . $requestProvider->getCookieParam("token"));

        $xsrfToken = $requestProvider->getParam("X-CSRF-TOKEN");
        if ($xsrfToken != $token["xsrfToken"]) {
            throw new Exception("X-CSRF-TOKEN is not valid.", 401);
        }

        if ($token) {
            return $this->response(static::class, "Delete object with id $id.");
        }
        $this->error("Token $token is invalid.", 401);
    } catch (Exception $e) {
        throw new Exception($e->getMessage(), 400);
    }
}
```

Die Webapplikation lehnt den Request ab, da das CSRF-Token nicht gefunden werden konnte. Auch wenn der Angreifer, den Header setzen könnte, müsste er erst in den Besitz des korrekten CSRF-Tokens kommen. Da er aber nicht auf den Localstorage zugreifen kann, wird dies ebenfalls schwierig.