UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Praktikum 6

Websicherheit

Zu bearbeiten bis zum 06. Juni 2023

Cybersecurity im Sommersemester 2023

Jan Wichelmann, Anja Köhl

Einführung

Dieses Praktikum behandelt verschiedene Aspekte der Websicherheit, mit Fokus auf die Folgen einer mangelhaften Validierung von Benutzereingaben.

Bei den ersten drei „klassischen“ Angriffen geht es um die ungewollte Einbettung und Ausführung von JavaScript (Cross-Site-Scripting (XSS), JSON Injection). Anschließend betrachten wir noch zwei andere, etwas weniger bekannte Angriffstechniken, die nichtsdestotrotz mindestens so schwere Folgen haben können wie ein XSS-Angriff.

Der XSS-Blog

Bei Ihrem letzten Streifzug durch das Web haben Sie einen kleinen selbstgeschriebenen Blog gefunden: `https://teaching.its.uni-luebeck.de/cs/lab/web-xssblog`.

Dieser erlaubt es Besuchern, unter jedem Beitrag einige Kommentare zu hinterlassen. Vor allem aber scheint der Blog bestens dazu geeignet zu sein, Alice endlich die Gefahr durch bösartige Skripte nahezubringen und sie davon abzuhalten, ständig auf alle möglichen Links zu klicken. Hierzu bitten Sie Alice, einige Geheimnisse in ihren Cookies abzulegen, die Sie mit verschiedenen Tricks stehlen möchten.

Die Blogseite ist denkbar einfach konstruiert: Sie können nach erfolgtem Login beliebig viele Blogbeiträge erzeugen, wobei diese je nach ausgewähltem Angriffstyp verschiedene Schwächen bei der Filterung von Eingaben aufweisen. Das Ziel der XSS-Aufgaben ist jeweils, Alice dazu zu bringen, ihre Cookies (also den Wert des `document.cookie`-Objekts) als Kommentar zu posten.

Hierbei hilft Ihnen die folgende Funktion:

```
function createNewComment(blogEntryId, name, email, text)
{ ... }
```

Diese Funktion erstellt für die angegebene Blog-ID (in der Adressleiste ablesbar) einen neuen Kommentar, der den angegebenen Namen, die angegebene E-Mail-Adresse und den angegebenen Text aufweist. Die Funktion wird ursprünglich nur von dem Eingabeformular unter einem Blogbeitrag verwendet, lässt sich jedoch auch auf andere Weise aufrufen. Ein Beispiel:

```
// Erstellt einen Kommentar von "Alice" unter dem Blogbeitrag w8kckcnnyy
createNewComment("w8kckcnnyy", "Alice", "alice@alice.al", "Toller Blogbeitrag!");
```

Zusätzlich gibt es zwei Buttons „Aktualisieren“ und „Alice informieren“. Ersterer aktualisiert die angezeigten Kommentare, um gegebenenfalls neue Kommentare von Alice darzustellen (alternativ können Sie auch einfach die ganze Seite neu laden). Zweiterer schickt an Alice eine Nachricht mit der aktuellen Blogseite. Alice besucht daraufhin die Seite, und fällt dann auf Ihre Angriffstechnik herein. Wenn Sie erfolgreich sind, generiert Alice' Besuch einen neuen Kommentar, der ihre Cookies enthält.

Technischer Hinweis: Selbstverständlich haben wir für das Praktikum keine Alice zu einer Woche Browser-Dienst verpflichtet. Stattdessen simulieren wir ihren Browser mithilfe einer einfachen JavaScript-Umgebung. Diese enthält das Cookie-Objekt und die `createNewComment`-Funktion, baut jedoch kein vollständiges DOM auf. Abgesehen von der spezifischen Angriffsstrategie sind sich die Exploits sehr ähnlich, und benötigen keinerlei Zugriff auf andere Funktionen und Komponenten.

Aufgabe 1 Reflected XSS

Alice klickt sehr gern auf Links. Sie haben ihr bereits erfolgreich beigebracht, nur auf Links zu klicken, die mit einer vertrauenswürdigen Domain beginnen, aber leider achtet sie nach wie vor nicht darauf, wie der Rest des Links zusammengesetzt ist. Wenn Sie also unter einem Blogbeitrag einen Kommentar mit einem Link darin platzieren, der auf die vertrauenswürdige `teaching`-Subdomain vom ITS zeigt, wird Alice diesen garantiert aufrufen.

1. Generieren Sie einen neuen Blogbeitrag vom Typ *Reflected XSS*. Sie werden im Anschluss automatisch zu dem neu erstellten Beitrag weitergeleitet.
2. Analysieren Sie die Adresse, die Ihnen der Browser in der Adressleiste anzeigt. Was passiert, wenn Sie den `id`-Parameter so verändern, dass dieser auf einen nicht existenten Blog-Beitrag verweist? Welchen Einfluss hat der `title`-Parameter?
3. Versuchen Sie, im `title`-Parameter einfache HTML-Tags zu verwenden, beispielsweise `<b>Fetter Text</b>`. Wie verändert sich die Ausgabe?
4. Verwenden Sie nun statt `<b>Fetter Text</b>` die JavaScript-Tags `<script>alert(42);</script>`. Welches Verhalten beobachten Sie?
5. Erstellen Sie eine URL, die mithilfe der Funktion `createNewComment` einen Kommentar unter einem Blogbeitrag anlegt. Modifizieren Sie dann die URL, um als Kommentartext den Wert des Cookie-Objekts Ihres Browsers zu verwenden.
6. Posten Sie Ihre präparierte URL in codierter Form als Kommentar unter dem zu Beginn erstellten Blogbeitrag, und rufen Sie Alice an. Aktualisieren Sie nach einigen Sekunden die Kommentarliste.

   Falls Sie keinen Fehler gemacht haben, müsste Alice einen neuen Kommentar angelegt haben, der ihre Cookies und damit auch die gesuchte geheime Zeichenkette enthält. Geben Sie diese Zeichenkette auf dem Praktikumsserver ein. ◀ 90 / 0

Hinweise:

- Da URLs weder Leerzeichen noch bestimmte Sonderzeichen enthalten dürfen, werden diese im Vorfeld codiert. Die meisten Browser sollten es Ihnen ohne Weiteres erlauben, beliebige Strings mit Leerzeichen in der Adressleiste einzugeben. Diese werden dann beim Senden des HTTP-Anfrage im Hintergrund codiert und an den Server geschickt. Wenn Sie die URL aus der Adressleiste kopieren, sollte der Browser diese ebenfalls codieren und so in der Zwischenablage ablegen. Sie können feststellen, ob eine URL codiert ist, indem Sie beispielsweise schauen, ob Leerzeichen als `%20` codiert sind.

  Falls Sie eine URL manuell codieren wollen, könnte Ihnen die `encodeURI`-Funktion[1] von JavaScript weiterhelfen.

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURI`

Aufgabe 2 Stored XSS

Alice klickt nicht nur gern auf Links, sondern verschickt auch sehr gern E-Mails. Noch besser ist die Kombination aus beidem: Ein Link, der ein Fenster zum Verschicken einer E-Mail öffnet! Natürlich ist sie vorsichtig und schaut, dass die im Browser angezeigte Link-Adresse ein korrektes Format hat, also zum Beispiel

```
mailto:alice@uni-luebeck.de
```

Dort einfach ein `<script>`-Tag unterzubringen würde also auffallen, und sowieso nicht allzu viel bewirken. Folglich ist ein neuer Ansatz nötig.

Um E-Mail-Adressen für die Darstellung „sicher“ zu machen, hat der Programmierer eine Funktion `htmlspecialchars` eingebaut, die bestimmte HTML-Steuerzeichen entfernt; das betrifft die Zeichen `<`, `>`, `&` und `&quot;`. Also ist alles in Ordnung, oder?

1. Finden Sie heraus, welche Zeichen benutzt werden können, um in JavaScript Strings darzustellen.
2. Beim Aktualisieren der Kommentare wird der Link für eine E-Mail an den Absender im JavaScript-Code folgendermaßen erzeugt:

   ```
   emailLink.innerHTML = `<a href='mailto:${email}'>(icon)</a>`;
   ```

   Der Parameter `${email}` enthält dabei die mit `htmlspecialchars` verarbeitete E-Mail-Adresse.

   Können Sie eine Eingabe für das E-Mail-Adress-Feld konstruieren, die in das `<a ...>`-Tag ein neues Attribut `onclick` mit Inhalt `alert(1);` einfügt? Was passiert, wenn Sie anschließend auf den daraus erzeugten E-Mail-Link klicken?

   Hinweis: Der Ansatz hier ist sehr ähnlich zu den SQL-Injections.

3. Konstruieren Sie eine Eingabe für das E-Mail-Adress-Feld, sodass beim Klicken auf den erzeugten E-Mail-Link ein neuer Kommentar mit Ihren aktuellen Cookies generiert wird. ◀ 90 / 0

   Informieren Sie anschließend Alice. Nicht vergessen: Alice klickt nur auf E-Mail-Links, bei denen eine sinnvolle E-Mail-Adresse angezeigt wird!

Aufgabe 3 JSON Injection

Viele Webseiten implementieren eine dynamische Weboberfläche mittels einer Web API. Eine solche API besteht aus einer Reihe von Adressen, die vom in der Webseite laufenden JavaScript asynchron und im Hintergrund abgefragt werden, um die gezeigte Oberfläche dynamisch mit Daten anzureichern. Das gleiche passiert auch in unserer Blog-Software: Wenn Sie auf „Aktualisieren“ drücken, wird nicht die komplette Seite neu geladen, sondern es wird im Hintergrund eine aktuelle Liste aller Kommentare abgefragt und damit dann die Kommentarspalte befüllt.

Als Datenstruktur für die Kommunikation mit Web APIs hat sich JSON (*JavaScript Object Notation*) etabliert, da es in diesem Szenario effizienter einsetzbar ist als andere klassische Formate wie XML. Ein großer Vorteil von JSON ist, dass es – wie der Name schon sagt – unmittelbar der Syntax für Objekte in JavaScript entspricht, es kann also theoretisch mit der `eval`-Funktion direkt als JavaScript-Code ausgeführt werden. Hier ist ein Beispiel für ein JSON-Objekt, das beispielsweise aus einer Messaging-App stammen könnte:

```
{
  "name": "Alice",
  "accountId": 42424242,
  "devices":
  [
    {
      "type": "Phone",
      "verified": true
    },
    {
      "type": "Notebook",
      "verified": false
    }
  ]
}
```

Ein JSON-Objekt ist somit eine Sammlung von Schlüssel/Wert-Paaren, und wird mit geschweiften Klammern markiert. Listen sind mit eckigen Klammern umschlossen. JSON unterstützt nativ nur Strings, Zahlen, `true`, `false` und `null`. Dazu gibt es keinerlei Typsicherheit, das heißt eine Liste kann Elemente verschiedensten Typs enthalten, und Objekte und Listen können beliebig tief geschachtelt werden.

In dieser Aufgabe nutzen Sie die große Nähe von JSON zu JavaScript und eine mangelhafte Filterung von Benutzereingaben aus, um Alice unbemerkt beliebigen Code unterzuschieben.

1. Nutzen Sie die Entwicklertools Ihres Browsers, um die HTTP-Anfrage beim Aktualisieren der Kommentare aufzuzeichnen, und machen Sie sich mit dem Format der JSON-Antwort vertraut.

   Erstellen Sie anschließend einen Kommentar unter dem Namen `alice"`, und betrachten Sie erneut die JSON-Antwort beim Aktualisieren der Kommentare. Was fällt Ihnen auf?

   Versuchen Sie, einen zusätzlichen Schlüssel `"x"` mit dem Wert `42` in die JSON-Antwort einzubetten.

   Hinweise:

   - Um versehentliche Endlosschleifen zu verhindern, wird die Kommentarliste beim Erstellen eines Kommentars im „sicheren Modus“ (`?secure=true`) geladen, bei dem keine Exploits möglich sind. Nur beim Klicken des „Aktualisieren“-Buttons wird der unsichere Modus (`?secure=false`) verwendet. Vergleichen Sie die JSON-Antworten für beide Fälle, um den Unterschied festzustellen.
   - Falls Sie versehentlich ungültiges JSON erzeugen und die Kommentarliste nicht mehr im unsicheren Modus geladen werden kann, erstellen Sie einfach einen weiteren Kommentar mit „normalen“ Eingaben. Dann wird die Liste im sicheren Modus geladen, und Sie können im Anschluss die fehlerhaften Kommentare löschen.
   - Auch hier ist die Idee wieder sehr ähnlich zu den SQL-Injections.

2. Anstatt der langweiligen und spitzfindigen `JSON.parse`-Funktion nutzt die Blog-Software die `eval`-Funktion, um das JSON-Objekt aus dem Response-String zu extrahieren. Die `eval`-Funktion interpretiert den Response-String einfach als JavaScript-Quelltext, und erstellt bei dessen Ausführung entsprechend ein Objekt.

   Experimentieren Sie etwas mit der `eval`-Funktion, und geben Sie die folgenden Befehle nacheinander in der Browser-Entwicklerkonsole ein:

   ```
   eval('var x = {"a": "abc", "b": [1, "xy", { "c": 1, "d": false }]}')
   x
   eval('(function() { return 42; })')
   eval('(function() { return 42; })()')
   eval('var x = { "a": 21+21 }')
   x
   eval('var x = { "a": (function() { return 42; })() }')
   x
   ```

   Welche Rückgabewerte und welches Verhalten beobachten Sie jeweils? Versuchen Sie, Ihre Beobachtungen zu erklären.

   Hinweis: Informieren Sie sich über Funktionen in JavaScript, und speziell über anonyme Funktionen[2].

3. Nutzen Sie die Erkenntnisse aus den beiden vorangegangenen Aufgabenteilen, um eine Eingabe für das „Name“-Feld zu konstruieren, die beim Aktualisieren der Kommentarliste einen neuen Kommentar mit dem Wert des Cookie-Objekts als Text erzeugt. Lassen Sie anschließend Alice die Seite aufrufen.

◀ 100 / 0

Aufgabe 4 Overposting (freiwillig)

Bei der Verarbeitung von Benutzereingaben benutzen viele Web APIs sogenanntes *Model Binding*. Bei dieser Technik wird die notwendige Deserialisierung und Validierung der übertragenen (JSON-)Daten vom zugrundeliegenden Framework teilweise abstrahiert, sodass der Entwickler der API selbst nur mit Objekten seiner jeweiligen Programmiersprache arbeiten muss.

Ein Beispiel: In C# mit dem ASP.NET Core-Framework erstellt man zuerst eine Klasse, die die zu erwartenden Parameter und die gewünschten Formate festlegt (auch *Model* genannt):

[2] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function`

```
public class Movie
{
    public int Id { get; set; }

    [Required(ErrorMessage = "Please specify a name.")]
    public string Name { get; set; }

    [Required(ErrorMessage = "Please specify a year.")]
    [Range(1900, 2100, ErrorMessage = "The year must be between 1900 and 2100.")]
    public int Year { get; set; }
}
```

Ein `Movie`-Objekt beschreibt einen Film, der eine (interne) ID, einen Namen und ein Veröffentlichungsjahr hat. In den eckigen Klammern über den jeweiligen Eigenschaften sind die Validierungsanweisungen spezifiziert. So sind sowohl der Name als auch das Jahr verpflichtende Angaben, wobei das Jahr zwischen 1900 und 2100 liegen muss.

Wenn man einen neuen Film anlegen möchte, schickt man beispielsweise per HTTP POST eine Anfrage an den API-Endpunkt `/movies/create`:

```
POST https://its-movie-database.com/movies/create

{"name":"Stargate","year":1994}
```

Das Framework decodiert nun das empfangene JSON-Objekt, und wendet die in der Klasse definierten Validierungsanweisungen an. Wenn die Validierung fehlschlägt (weil das Jahr beispielsweise 2200 beträgt), gibt der Server automatisch einen Fehler `400 Bad Request` zurück. Ansonsten gibt das Framework das decodierte `Movie`-Objekt an die eigentliche Anwendungslogik (den *Controller*) weiter:

```
[HttpPost("movies/create")]
public IActionResult CreateMovie([FromBody] Movie movie)
{
    // Store movie somewhere...

    // HTTP 200 OK
    return Ok();
}
```

Bei der Abfrage einer Liste von Filmen gibt die Anwendungslogik lediglich ein `List<Movie>`-Objekt zurück, das dann vom Framework wieder zu JSON serialisiert und an den Client geschickt wird:

```
{
    "movies": [
        {"id": 1, "name": "Stargate", "year": 1994},
        {"id": 2, "name": "Star Trek", "year": 1979}
    ]
}
```

Auf diese Weise muss sich der Entwickler einer API nicht mit Details wie dem Übertragungsformat von Daten beschäftigen, und durch die sehr einfachen und kurzen Validierungsanweisungen wird viel redundanter und fehleranfälliger Code gespart. Folglich ist *Model Binding* generell ein aus Security-Sicht empfehlenswertes Konzept, das den Anwendungscode vereinfacht und viele anfällige Verarbeitungsschritte in das gut getestete Framework verlagert.

Dennoch sind auch hier bestimmte Angriffe möglich. Einmal sind die standardmäßig angebotenen Validierungsanweisungen semantisch beschränkt (im obigen Beispiel würde das Framework ohne Probleme einen Namen akzeptieren, der nur aus einem Punkt besteht – aber als Filmname ergibt das keinen Sinn), und müssen daher in vielen Fällen durch weitere manuelle Validierungsschritte ergänzt werden, um die Eingabe ungültiger Daten zu verhindern. In dieser Aufgabe geht es jedoch um einen anderen Angriff, das sogenannte *Overposting*.

Beim Overposting nutzt der Angreifer aus, dass das Model Binding standardmäßig alle Eigenschaften setzt, deren Namen im Request vorkommen, auch wenn das vom Entwickler gar nicht beabsichtigt war. So ließe sich das obige Beispiel leicht modifizieren, um auch eine ID mitzusenden:

```
POST https://its-movie-database.com/movies/create

{"id":42,"name":"Stargate","year":1994}
```

Diese ID sollte aber eigentlich vom Datenbanksystem vergeben werden! Wenn beim Einfügen des neuen Objekts in die Datenbank kein weiterer Validierungsschritt erfolgt, kann es hier möglicherweise zu einem Konflikt mit einem bereits existierenden Film kommen, der die ID 42 hat.

Zurück zum Blog: Wenn Sie einen neuen Blogbeitrag anlegen, enthält dieser einen automatisch generierten und speziell hervorgehobenen Kommentar von „Admin“. Offenbar ist also auf Clientseite Logik vorhanden, um bestimmte Kommentare hervorgehoben darzustellen, aber keine Möglichkeit, selbst solche Kommentare zu erstellen. Oder vielleicht doch?

1. Analysieren Sie mithilfe der Entwicklertools Ihres Browsers die JSON-Antwort beim Abfragen der Kommentarliste, und die übertragenen JSON-Daten beim Erstellen eines neuen Kommentars. Was fällt Ihnen auf?
2. Versuchen Sie manuell eine Anfrage zu konstruieren, die einen hervorgehobenen Kommentar erstellt. Sie können dies entweder über ein JavaScript-Kommando bewerkstelligen (als Vorlage ist die Implementierung der Funktion `createNewComment` in `main.js` gut geeignet), oder durch Modifizieren einer älteren Anfrage zum Erstellen eines Kommentars. Einige Browser (z.B. Firefox) erlauben es, eine bereits gesendete HTTP-Anfrage zu bearbeiten und erneut zu senden.

   Wenn es Ihnen gelingt, einen hervorgehobenen Kommentar zu erstellen, wird das System einen weiteren Kommentar mit dem geheimen String zur Einreichung auf dem Praktikumsserver generieren. ◁ 100 / 0

Aufgabe 5 Privilege Escalation (freiwillig)

Nach all Ihren Experimenten mit Sicherheitslücken möchten Sie Ihre Spuren verwischen, und alle Hinweise auf Ihre Aktivitäten löschen. Leider bietet die Benutzeroberfläche keinerlei offensichtliche Möglichkeit, um Blogbeiträge zu löschen, Sie müssen sich also erneut auf die Suche nach Sicherheitslücken machen.

Ein weiteres klassisches Sicherheitsproblem bei Webapplikationen sind *ungeschützte Routen*. Eine Route ist ein URL-Bestandteil, der eine bestimmte Aktion in einer Webapplikation ansteuert. So wird bei einem HTTP GET an die Route

```
/blog/show
```

mit einem Parameter `?id=<id>` eine HTML-Seite mit dem Blogbeitrag mit der ID `<id>` zurückgegeben. Ein HTTP POST an die Route

```
/blog/<id>/comments
```

führt hingegen zur Erstellung eines Kommentars unter dem Blogbeitrag mit der ID `<id>`, und liefert eine JSON-Antwort zurück. Wie Sie sehen, korrespondiert die URL hier nicht wie gewohnt zu einer physischen Datei (wie z.B. `/js/main.js`), sondern wird von der Applikation verarbeitet.

Während der Entwicklung einer Applikation werden oft temporäre Routen angelegt, um beispielsweise die Daten aller Benutzer abzufragen oder sich einen internen Zustand ausgeben zu lassen. Zusätzlich gibt es noch versteckte Routen, die für die Administration vorgesehen und für andere Benutzer nicht sichtbar sind. Während bei letzteren gewöhnlich auf eine strikte Zugriffskontrolle geachtet wird, kann dies bei Debugging-Routen ausbleiben, da diese selten für den Produktiveinsatz gedacht sind, und vor dem Deployment entfernt werden. Der einzige Schutz bei vergessenen Debugging-Routen ist also die Tatsache, dass ein Angreifer diese nicht kennt – Security by Obscurity.

Hinweis: Diese Aufgabe ist etwas anders als die vorherigen Aufgaben, da es hier nicht primär um die syntaktisch korrekte Umsetzung einer bestimmten Angriffstechnik mit klaren Zielen geht. Stattdessen führt hier geschicktes Raten und Ausprobieren zum Ziel – also die Methode, die die Grundlage für viele reale Angriffe ist.

1. Versuchen Sie, in der Blogsoftware versteckte Entwickler/Debugging/Admin-Routen zu identifizieren. Arbeiten Sie sich dann weiter vor, bis Sie in der Lage sind einen Blogbeitrag zu löschen.

   Hinweise:

   - Falls Sie mit Raten nicht weiterkommen sollten, lohnt sich vielleicht ein Blick in den Seitenquelltext und/oder von der Seite verwendete Skriptdateien. Manchmal gibt es dort Hinweise auf bereits entfernte Codefragmente, oder Funktionen die auf der Website nicht verwendet werden.
   - Die Basis-URL für die Blogsoftware ist:

```
https://teaching.its.uni-luebeck.de/cs/lab/web-xssblog/<route>
```

Der Platzhalter `<route>` wird also durch die jeweilige Route ersetzt.

- Wenden Sie sich gern an die Betreuer, falls Sie ein beobachtetes Verhalten nicht erklären können oder einen Denkanstoß benötigen.

2. Geben Sie den beim Löschen eines Blogbeitrags angezeigten Code auf dem Praktikumsserver ein. ◁ 100 / 0