Solutions to Selected Problems

Guide to Internet Cryptography

Companion Material

February 24, 2026

Preface This document provides solutions to selected problems from the book *Guide to Internet Cryptography: Security Protocols and Real-World Attack Implications*. The material is intended for educational use in courses and self-study.

Book website: https://link.springer.com/book/10.1007/978-3-031-19439-9

1 Chapter 20: Web Security and Single Sign-On Protocols

Problem 20.1 URLs

Consider the following URL:
https://en.wikipedia.org/wiki/URL#Internationalized_URL

(a) How are the destination IP address and TCP port number determined?

(b) Which part of this URL will be sent to the web server when issuing a GET request for this resource?

(c) Which part of this URL is only used in the browser and is never transmitted? Which purpose does this part have?

Solution

(a) The destination IP address is determined by sending a DNS query for `en.wikipedia.org`. The destination TCP port is the well-known port 443 associated with the protocol `https`.

(b) `wiki/URL`

(c) `#Internationalized_URL`. This part is just a pointer to an achor element contained in the HTML document received as a response to the GET request.

Problem 20.2 Same Origin Policy

Suppose the web page `https://secure.banking.com` embeds a JavaScript library from `https://xsscheats.org`. Will this library be able to access the password of an online banking customer?

Solution

Yes. JavaScript code that is embedded in a web page somehow 'inherits' the web origin of this web page. Here the folklore version of the Same Origin Policy does not apply ("different web origins imply no mutual access"), since web developers assume that any JavaScript code embedded in a web page, even if loaded from a different web origin, must be trustworthy. This feature is frequently used in web applications when loading large JavaScript libraries from CDN servers.

Problem 20.3 Cross-origin communication

You are a car salesman for a Korean car manufacturer, and you are running your car database at https://salesguy.co.uk. You can retrieve a configuration list from this database to configure the car your customer wants to purchase. Now you have to transfer the completed configuration to https://koreancars.kr/orders. Please sketch an HTML form that can do this.

Solution

```
<!DOCTYPE html>
<html>
<head>
    <title>Submit Car Order</title>
</head>
<body>

<h2>Car Configuration</h2>

<form action="https://koreancars.kr/orders" method="POST">

    <!-- Customer information -->
    <label for="customer_name">Customer Name:</label>
    <input type="text" id="customer_name" name="customer_name" required>
    <br><br>

    <label for="email">Customer Email:</label>
    <input type="email" id="email" name="email" required>
    <br><br>

    <!-- Car configuration -->
    <label for="model">Model:</label>
    <select id="model" name="model">
        <option value="sedan">Sedan</option>
        <option value="suv">SUV</option>
        <option value="electric">Electric</option>
    </select>
    <br><br>

    <label for="color">Color:</label>
    <input type="text" id="color" name="color">
    <br><br>
```

```
    <label for="engine">Engine Type:</label>
    <select id="engine" name="engine">
        <option value="petrol">Petrol</option>
        <option value="diesel">Diesel</option>
        <option value="hybrid">Hybrid</option>
        <option value="electric">Electric</option>
    </select>
    <br><br>

    <!-- Hidden field containing serialized configuration -->
    <input type="hidden" name="config_id" value="ABC123456">

    <button type="submit">Submit Order</button>

</form>

</body>
</html>
```

Problem 20.4 Reflected XSS

Suppose your XSS filter replaces all occurrences of the string `<script` with the empty string. This filter parses the 3rd-party content only once, and the filtered content is placed in the `<body>` element. Define a string that will, after filtering, start an alert window with the text "Your XSS filter sucks!".

Solution

Suppose we have an injection point where 3rd-Party content is placed after filtering.

```
<html>
<body>
 /*injection point*/
</body>
</html>
```

Now assume that the string
`<<scriptscript>alert("Your XSS filter sucks")</script>`
is submitted by the attacker to the filter. The filter returns
`<script>alert("Your XSS filter sucks")</script>` ,
and this is placed at the injection point.

```
<html>
<body>
<script>alert("Your XSS Filter sucks")</script> /*injected*/
</body>
</html>
```

Problem 20.5 CSRF

You are an investment banker, and a new startup company is approaching you. They aim to prevent CSRF attacks using a "military-grade three-factor authentication." Should you

fund them?

Solution

No. This would only work if three-factor authentication is applied to *any* HTTP request, which would render the web application unusable. If this "military-grade three-factor authentication" is only applied at the first login, it will be translated in a session cookie, and CSRF will still work.

Problem 20.6 SSO

Consider the generic SSO flow from Figure 20.13. Why do SP and IdP communicate via the browser in steps (2) and (4)? Wouldn't it be easier if they communicated directly?

Solution

In this generic flow, the web browser always acts as an (active) client, and both servers always act as (passive) servers. This facilitates implementation. However, there are SSO flows where servers act as bot server and client, and there is direct server-to-server communication.

Problem 20.7 MS Passport

Consider the script snarf.js in Figure 20.15. This script is loaded from `alive.znep.com`. Why should this script be able to read the HTTP cookies from `passport.com/ppsecure/404please`?

Solution

Expoiting an XSS vulnerability in `register.passport.com/reg.srf`, the script is included here and 'inherits' ther web origin `(https,register.passport.com,443)`. The URL `passport.com/ppsecure/404please` has the same web origin, thus the script is allowed to access all DOM properties here, and `document.cookie` is one of these DOM properties. The paths (which are different) are not included in the web origin, so they don't matter.

Problem 20.8 OpenID

(a) Why must OpenID identities be URLs?

(b) Sketch how a MitM attacker can impersonate an arbitrary number of end users at SP.

Solution

(a) Because there must be a method to *retrieve* these identities, and URLs are the easiest was to do this – they are both worldwide unique, and describe a methode how and where to retrieve the data.

(b) The token t in Figure 20.16 is a *bearer token*. This means that any client in posession of this token can use it to authenticate to the target server.
If TLS is used, the MITM cannot directly retrieve t. Instead, he must run a fake Relying party, and will thus learn the token in step (9).
If an association is used, the attacker can intervene in step (4), which is not protected

against MITM attacks.

Problem 20.9 OAuth and OpenID Connect

What are the differences between Figures 20.17 and 20.18?

Solution

20.17 only describes how to get *authorization* tokens; in Figure 20.18, the possobility to retrieve *authentication* tokens ('ID Token') is added. Otherwise, the two flows are strikingly similar.