**Exploiting XSS via WebSockets**

Like any other unsanitized input, embedding messages from a WebSocket connection into a website can lead to Cross-Site Scripting (XSS). Suppose an attacker can send a malicious XSS payload to other users so that it is embedded into their browser's Document Object Model (DOM). In that case, there is a valid XSS attack vector.

We will not go into too much detail about exploiting XSS vulnerabilities in this section; refer to the [Cross-Site Scripting (XSS)](https://academy.hackthebox.com/module/details/103) module for more info.

**Code Review - Identifying the Vulnerability**

We will attack a chat web application that utilizes WebSockets connections; accessing it, we can see that it allows us to send messages to the admin user:

A white background with black text

Description automatically generated

Because this is a chat web application, we can assume that the messages we send are displayed in the admin's browser, potentially meeting one of the conditions for an XSS attack. Let us analyze the source code to determine whether the data we send with messages is being properly sanitized:

Code: python

to\_admin = queue.Queue()

to\_user = queue.Queue()

<SNIP>

@sock.route('/userws')

def userws(sock):

while True:

if not to\_user.empty():

msg = to\_user.get()

sock.send(msg)

msg = sock.receive(timeout=1)

if msg:

to\_admin.put(msg)

@sock.route('/adminws')

def adminws(sock):

while True:

if not to\_admin.empty():

msg = to\_admin.get()

sock.send(msg)

msg = sock.receive(timeout=1)

if msg:

to\_user.put(msg)

The code defines two WebSocket endpoints, one for the user at /userws and one for the admin at /adminws. Since they are functionally identical, let's analyze how the user WebSocket connection is implemented. There is a to\_admin queue for messages from the user to the admin. Any message sent by the user is added to the queue and subsequently sent to the admin via the WebSocket connection.

Since the client's browser initializes WebSocket connections, let us analyze the client-side JavaScript code in index.html as well (we should also examine the code in admin.html to check how the WebSocket connection is initialized from the admin user's perspective, but in this case, both are functionally the same):

Code: html

<script>

var form = document.getElementById("chatform");

form.addEventListener('submit', sendMessage);

const socket = new WebSocket('ws://' + location.host + '/userws');

socket.addEventListener('message', ev => {

log('Admin', ev.data);

});

function log (user, msg){

var today = new Date();

var time = today.getHours() + ":" + today.getMinutes() + ":" + today.getSeconds();

document.getElementById('chat').innerHTML += `<div class="chat-message clearfix"><div class="chat-message-content clearfix"><span class="chat-time">${time}</span><h5>${user}</h5><p>${msg}</p></div></div><hr>`;

}

function sendMessage(event){

event.preventDefault();

let msg = document.getElementById("msg").value;

socket.send(msg);

log('You', msg);

document.getElementById("msg").value = '';

}

</script>

The chat messages received from the WebSocket connection are passed to the log function, which is set via the addEventListener call on the socket variable. The log function shows that the chat message is added to the DOM via the innerHTML property without any sanitization; since no sanitization is applied, XSS is possible.

**Debugging the Code Locally**

Let us run the web application locally to open both sides of the chat application and check the messages; we first must install the Flask and flask-sock dependencies using pip, and to avoid the need to have root privileges to run the web application, we will change the port to an unprivileged port, such as port 8000. After running the web application, we can open the chat endpoints by accessing http://127.0.0.1:8000/ and http://127.0.0.1:8000/admin in separate browser tabs.

Sending a benign HTML tag, such as <strike>PoC</strike>, from the user account to the admin confirms the absence of sanitization, as the tag is successfully injected:

A white screen with text

Description automatically generated

**Note:** Running the code locally also allows us to find potential XSS vulnerabilities using a dynamic approach by inspecting the browser's DOM after receiving a message; for a powerful toolset for identifying vulnerabilities, this can be combined with static analysis of the client-side JavaScript code.

**Exploitation**

Now that we have a working PoC, we can work on more fruitful XSS payloads. As expected, when sending the typical XSS payload <script>alert(1)</script>, it gets displayed as an empty message from the admin user's perspective (because the script tag is invisible):

A white screen with text

Description automatically generated

However, there is no alert pop-up due to a security measure designed to prevent XSS attacks, as stated in the [HTML5](https://www.w3.org/TR/2008/WD-html5-20080610/dom.html#innerhtml0) specification: "script elements inserted using innerHTML do not execute when they are inserted". Fortunately, other XSS payloads use event handlers. The [Payload All The Things](https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/XSS%20Injection#common-payloads) repository contains plenty of them. Sending the following payload results in an alert pop-up in the admin's browser:

Code: html

<img src='x' onerror='alert(1)'>