🔍 Technical Deep Dive: Reverse Engineering School Portal Authentication

Document Purpose: This comprehensive guide walks through the complete process of reverse-engineering a web application's authentication flow using browser developer tools. We'll cover every step from initial inspection to final automation.

Tools Used:

- DevTools (Network tab, Elements tab, Console)
- Node-RED for automation
- Home Assistant for integration
- wget for reliable file downloads

Part 1: Initial Reconnaissance

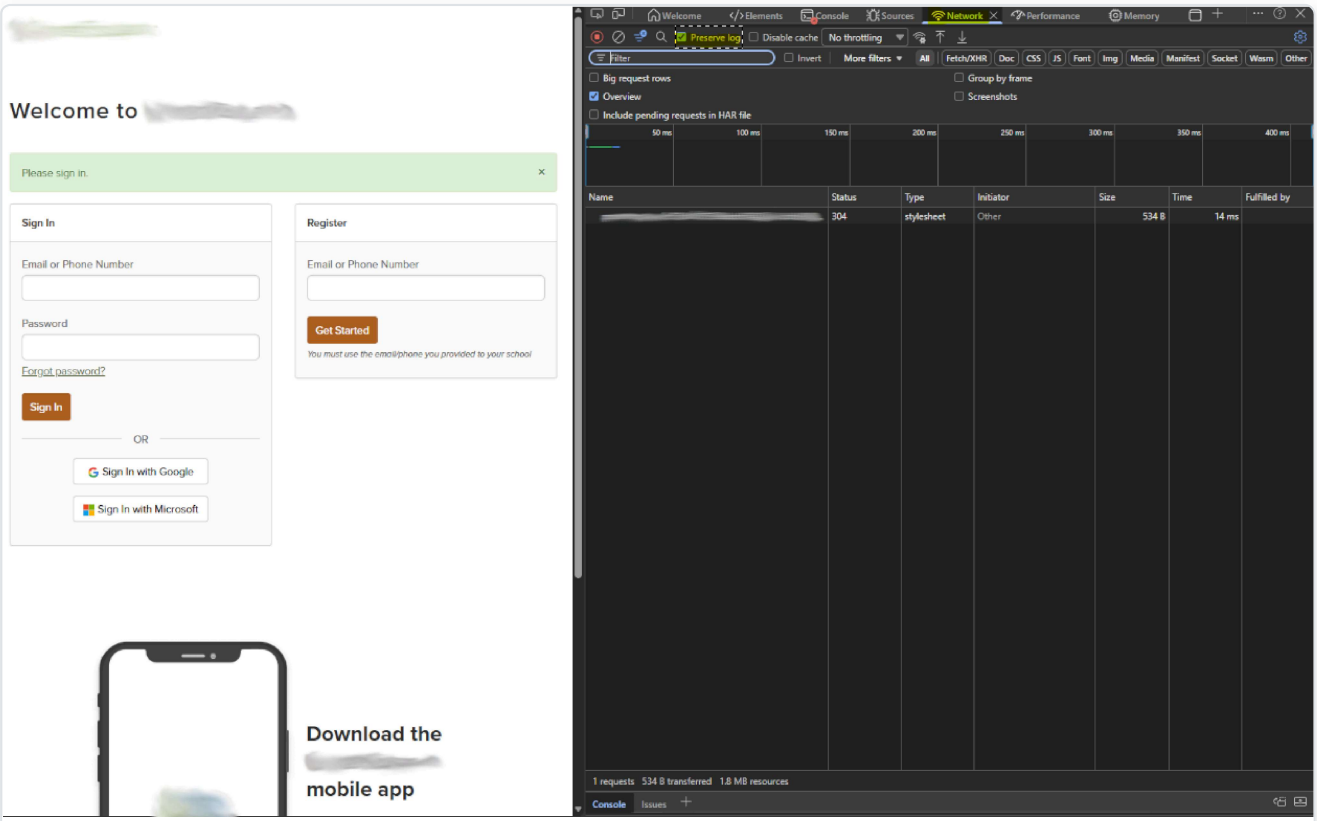
1 Open Browser Developer Tools

First, we need to see what's happening behind the scenes when logging into the portal.

Instructions:

1. Open the school portal login page in your browser
2. Press `F12` or `Ctrl+Shift+I` (Windows/Linux) or `Cmd+Option+I` (Mac)

3. Click the **Network** tab at the top
4. ✅ Check the **Preserve log** checkbox (very important!)



DevTools — Network tab open, Preserve Log checked, ready to capture requests

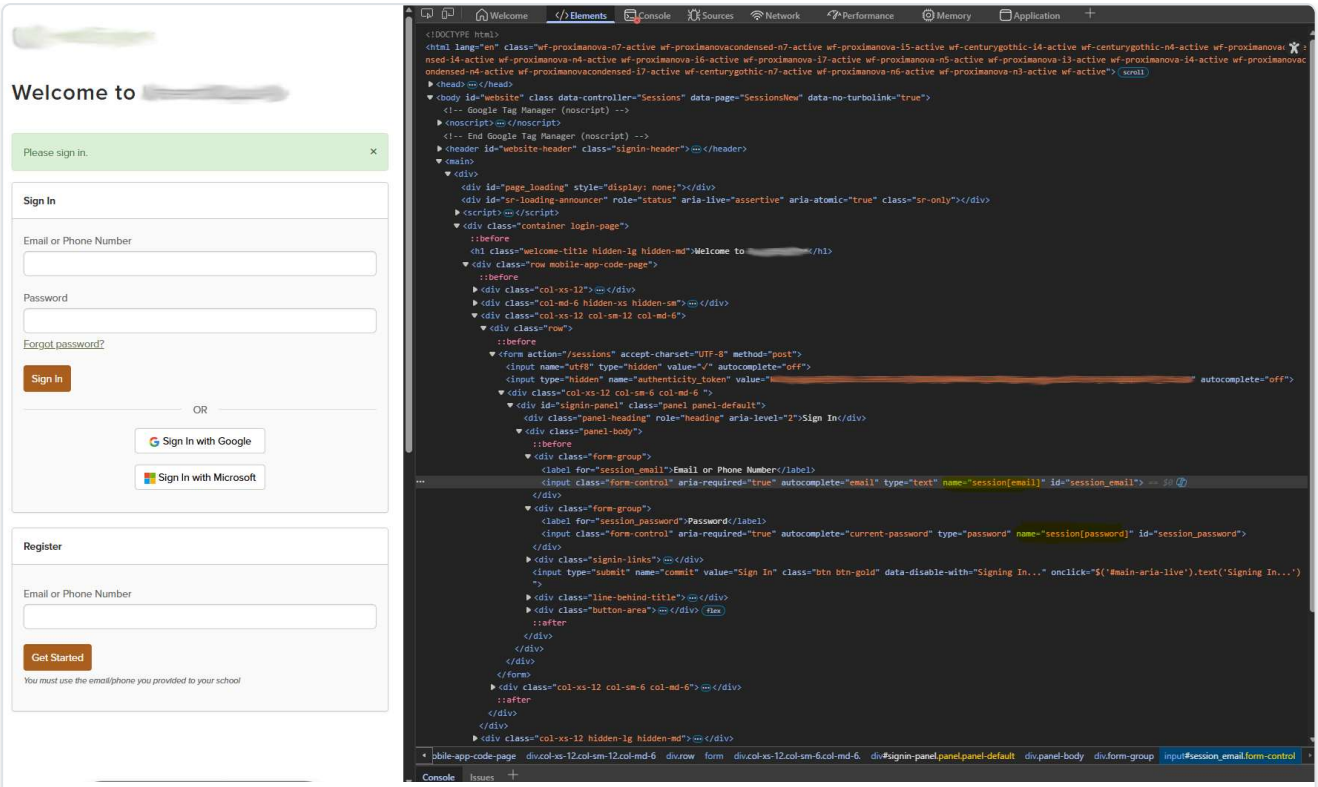
> 💡 **TIP:** When you submit a login form, the page often redirects. Without "Preserve log" enabled, you'll lose all the network requests and won't be able to see what happened!

2 Inspect the Login Form Fields

Before we submit anything, let's identify what data the form expects.

Finding the Email/Username Field:

1. Right-click on the **email/username input field** on the login page
2. Select **Inspect** (or **Inspect Element**)
3. The Elements tab will open and highlight the input field



Right-clicking the email field reveals the Inspect option — this jumps straight to that element in the Elements tab

What We Found:

HTML

```
<input type="email"
       name="session[email]"
       id="user_email"
       class="form-control"
       placeholder="Email address">
```

Key Information Extracted:

- `name="session[email]"` - This is the field name we need to submit in our POST request
- `type="email"` - HTML5 email validation
- `id="user_email"` - Could be used for automation/testing

Finding the Password Field:

Repeat the same process for the password field:

HTML

```
<input type="password"
       name="session[password]"
       id="user_password"
       class="form-control"
       placeholder="Password">
```

Key Information:

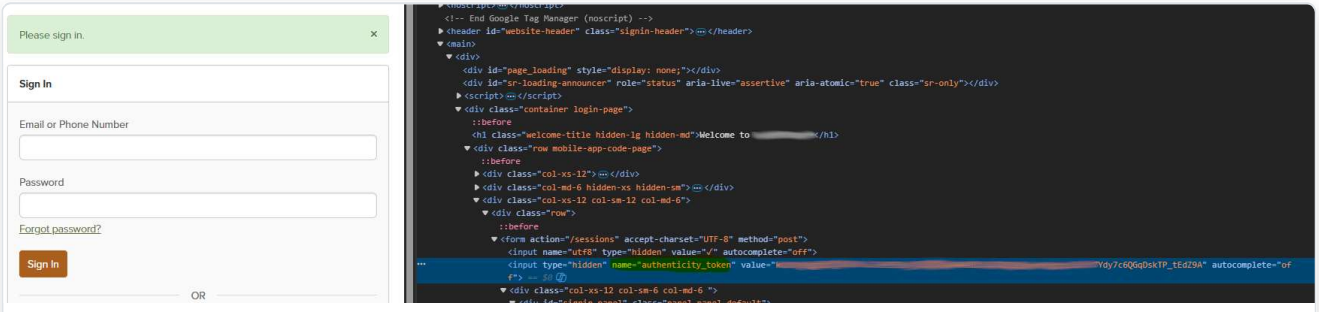
- `name="session[password]"` - Field name for password submission
- `type="password"` - Input is masked (dots/asterisks)

3 Discover the CSRF Token

Most modern web applications use CSRF (Cross-Site Request Forgery) tokens to prevent malicious login attempts. This is a critical security feature we need to handle.

Finding the Token:

1. In the Elements tab (still open from previous step), press `Ctrl+F` to open the search box
2. Search for: `authenticity_token`
3. Find the hidden input field containing the token



Searching for "authenticity_token" in the Elements tab reveals the hidden CSRF token input — this value must be extracted before every login attempt

What We Discovered:

HTML

```
<input type="hidden"
       name="authenticity_token"
       value="token">
```

> 🔴 **IMPORTANT:** This token is CRITICAL! Every login attempt needs this token, and it changes on every page load. We MUST extract this value before attempting to log in.

Why This Matters:

- The token proves we loaded the real login page from the server
- It prevents automated attacks from external websites
- It's tied to your current session
- It expires quickly (usually within minutes)

Verify It Changes:

1. Note the current token value
2. Refresh the page (press `F5`)
3. Inspect the form again
4. Compare - the token should be completely different!

> 💡 **TIP:** Copy the token value to a text editor temporarily so you can compare it after refresh. This confirms the token is dynamic and must be extracted fresh for each login attempt.

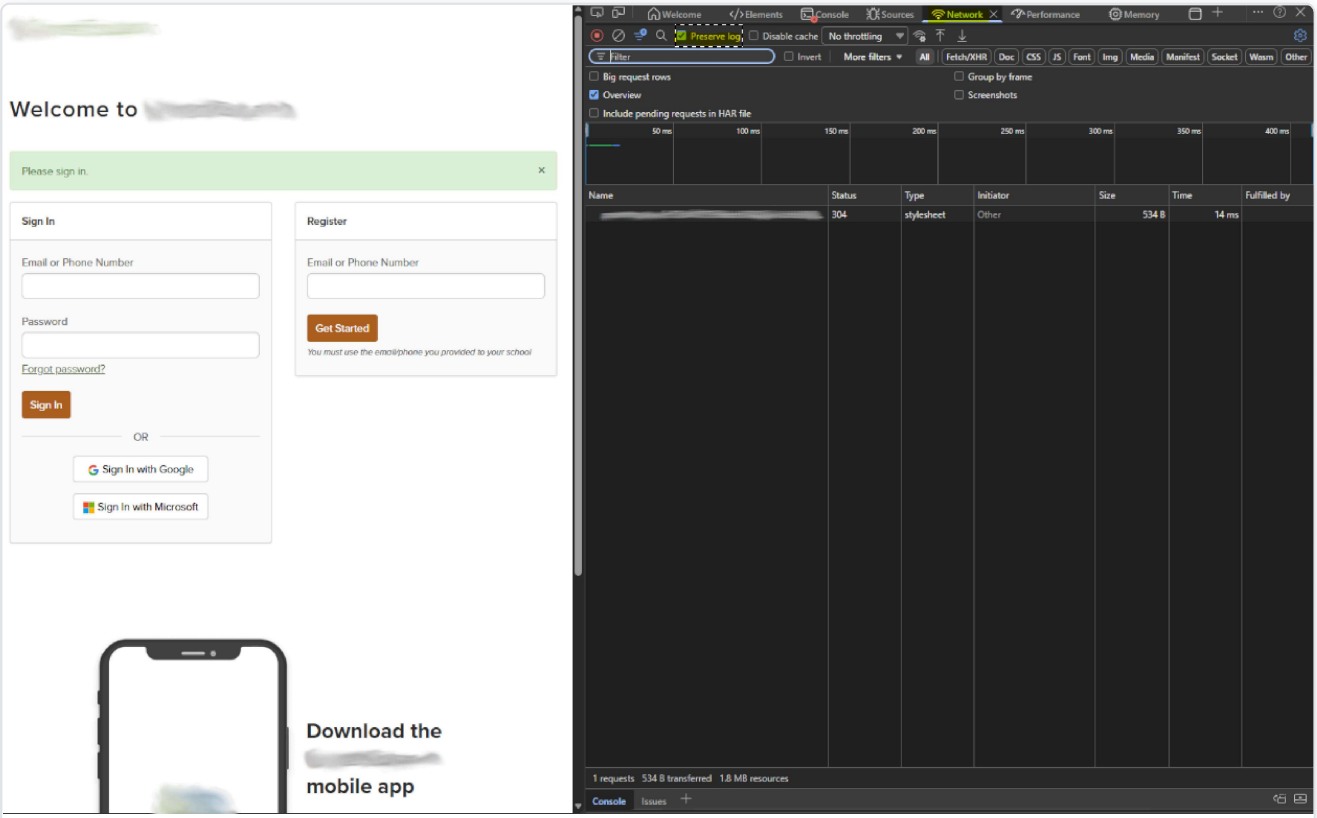
Part 2: Watching the Login Process Live

4 Prepare to Capture the Login Request

Now let's see what happens when we actually submit our credentials.

Setup:

1. Switch back to the **Network** tab in DevTools
2. Click the **Clear** button (🚫 circle icon) to remove old requests
3. Verify **Preserve log** is still checked ✅
4. Keep DevTools open and visible



Network tab cleared and ready — Preserve Log is checked so requests won't disappear when the page redirects after login

Execute Login:

1. Enter your credentials in the login form

2. Click the **Sign In** button

3. Watch the Network tab fill up with HTTP requests in real-time!

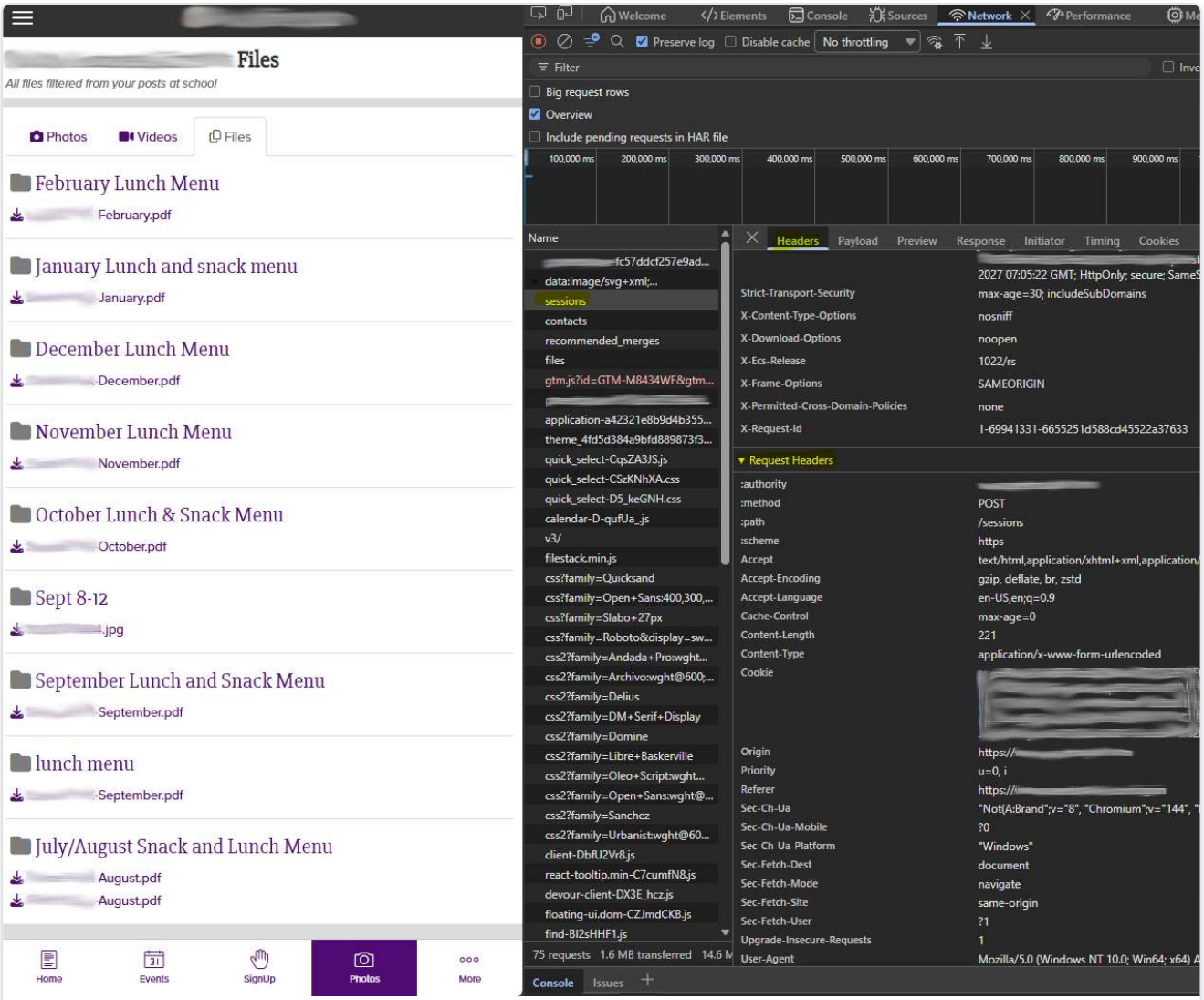
> 💡 **TIP:** Don't close DevTools! If you close it, you'll lose all the captured network traffic. Keep it open throughout the entire login process.

5 Identify the Critical Login Request

The Network tab now shows all HTTP requests. Let's find the one that actually performs the login.

What to Look For:

- **Method:** POST (shown in red/pink color or "POST" text)
- **Name:** Usually contains `sessions`, `sign_in`, `auth`, or `login`
- **Status:** 302 (redirect) or 200 (success)
- **Type:** document



The POST /sessions request appears in red — the 302 status means we're being redirected, which is exactly what a successful login looks like

9:03 AM 2/17/2026

In Our Case, We Found:

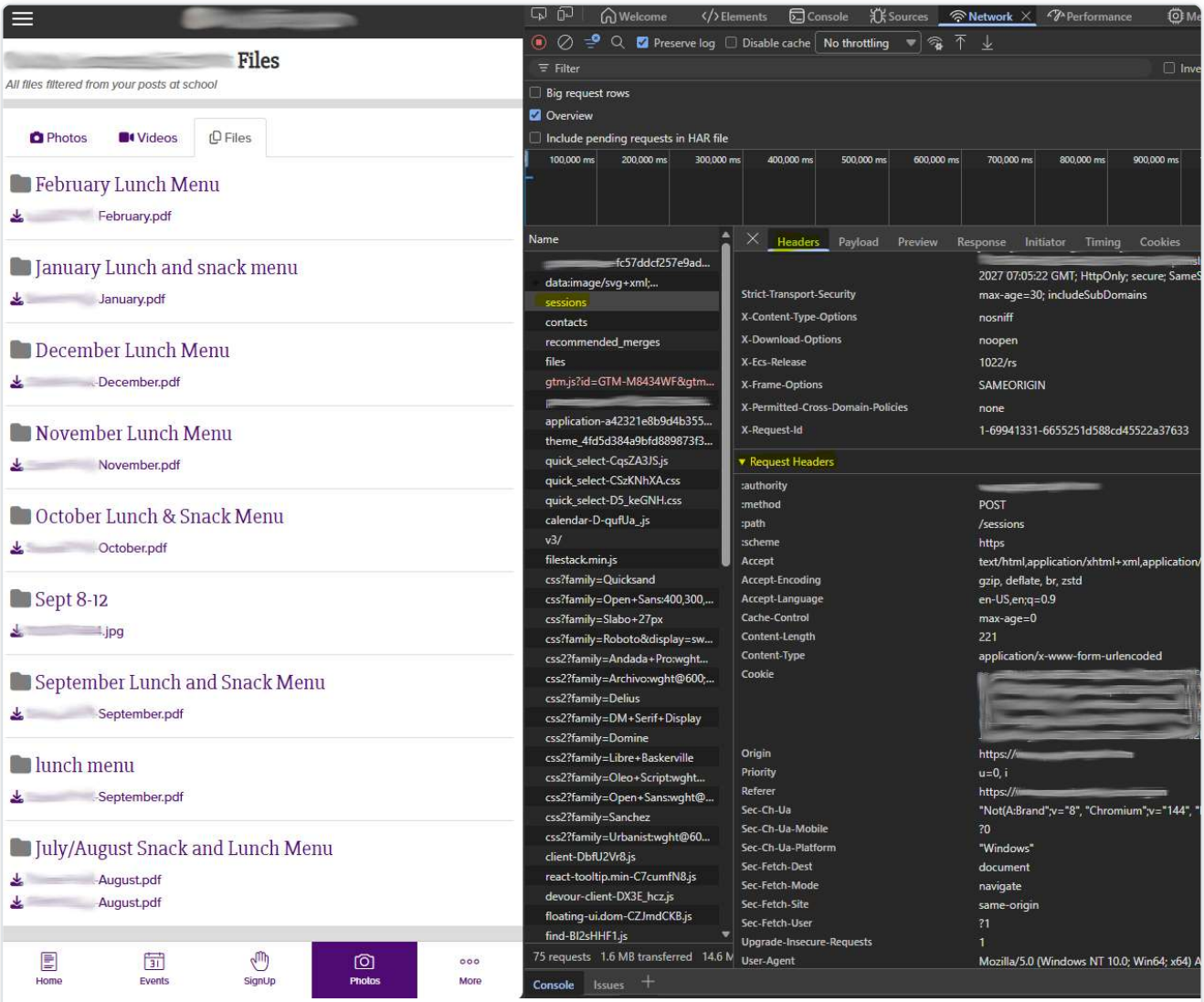
Property	Value	Meaning
Name	sessions	The login endpoint
Method	POST	Sending data to server
Status	302 (Found)	Redirect = Login succeeded!
Type	document	HTML response

> ✅ **SUCCESS:** The 302 status code is exactly what we want! It means we're being redirected after login, which typically indicates successful authentication.

6 Examine Request Headers in Detail

Click on the POST /sessions request to open the details panel.

Click on the **Headers** tab to see:



The Headers tab breaks down every header sent with the login request — Content-Type, Cookie, User-Agent, Origin, and Referer all need to be replicated in the automation

Request Headers Section:

HTTP Request

```
POST https://your-school-portal.com/sessions HTTP/1.1
Host: your-school-portal.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 213
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Cookie: ps_s=eWJhc2U2NGVuY29kZWRkYXRh...
Referer: https://your-school-portal.com/signin
Origin: https://your-school-portal.com
```

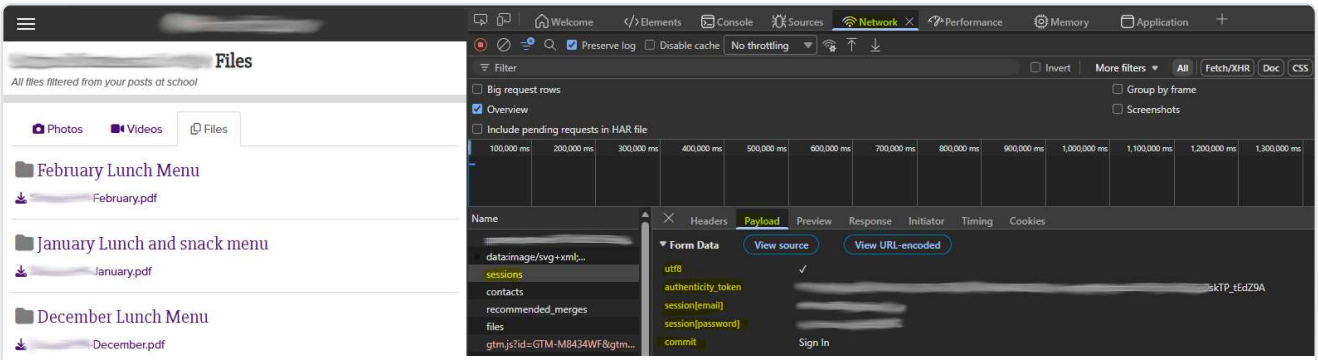
Critical Header Observations:

- **Content-Type:** `application/x-www-form-urlencoded` - Tells us the POST data format (like HTML form submission)
- **Cookie:** The signin page set a cookie (`ps_s`), and we're sending it back with our login request
- **User-Agent:** Full browser identification string - some servers check this!
- **Referer:** Shows we came from the signin page (security check)
- **Origin:** Prevents CSRF attacks from other domains

⚠️ **WARNING:** Many modern web applications will reject requests with missing or suspicious headers. We need to replicate ALL of these headers in our automated requests!

7 Inspect Form Data Payload

Scroll down in the Headers tab to find the **Request Payload** or **Form Data** section.



The Form Data section reveals every field sent with the POST — utf8, authenticity_token, session[email], session[password], and commit must all be included in the automated request

What We See Being Sent:

Form Data

```
utf8: ✓
authenticity_token: token
session[email]: [email protected]
session[password]: ••••••••••
commit: Sign In
```

Form Field Analysis:

- `utf8: ✓` - UTF-8 encoding marker (checkmark character)
- `authenticity_token` - The CSRF token we found earlier ✅
- `session[email]` - Matches the form field name exactly!
- `session[password]` - Password field (shown as dots in DevTools for security)
- `commit: Sign In` - The button text/value

> 🔴 **IMPORTANT:** This tells us EXACTLY what data to send in our Node-RED flow! Every field shown here must be included in our automated POST request.

8 Analyze the Login Response

Click on the **Response** tab to see what the server sent back.

Common Response Scenarios:

Scenario 1: HTML Redirect Page

HTML Response

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="refresh" content="0; url=/users/id/contacts">
  <title>Redirecting...</title>
</head>
<body>
  <p>You are being redirected...</p>
</body>
</html>
```

Note the URL in the meta refresh tag: `/users/id/contacts` - this is where successful logins go!

Scenario 2: 302 HTTP Redirect

Look at the **Response Headers** section:

HTTP Response Headers

```
HTTP/1.1 302 Found
Location: https://your-school-portal.com/users/id/contacts
Set-Cookie: ps_r=aGVyZV9pc19hbm90aGVyX2Nvb2tpZQ==; Path=/; Expires=...
Set-Cookie: ps_d=eWV0X2Fub3RoZXJfY29va2ll; Path=/; Expires=...
Set-Cookie: ps_s=dXBkYXRlZF9zZXNzaW9uX2Nvb2tpZQ==; Path=/; Expires=...
```

Critical Response Observations:

- **Location header:** Redirects to `/users/XXXXXX/` - this URL pattern means login succeeded!
- **Multiple Set-Cookie:** Three new cookies are being set - these are our authentication credentials!
- **Cookie names:** `ps_r`, `ps_d`, `ps_s` - all three are needed
- **Expiration:** These cookies are valid for extended periods (often months or years)

How to Verify Success vs Failure:

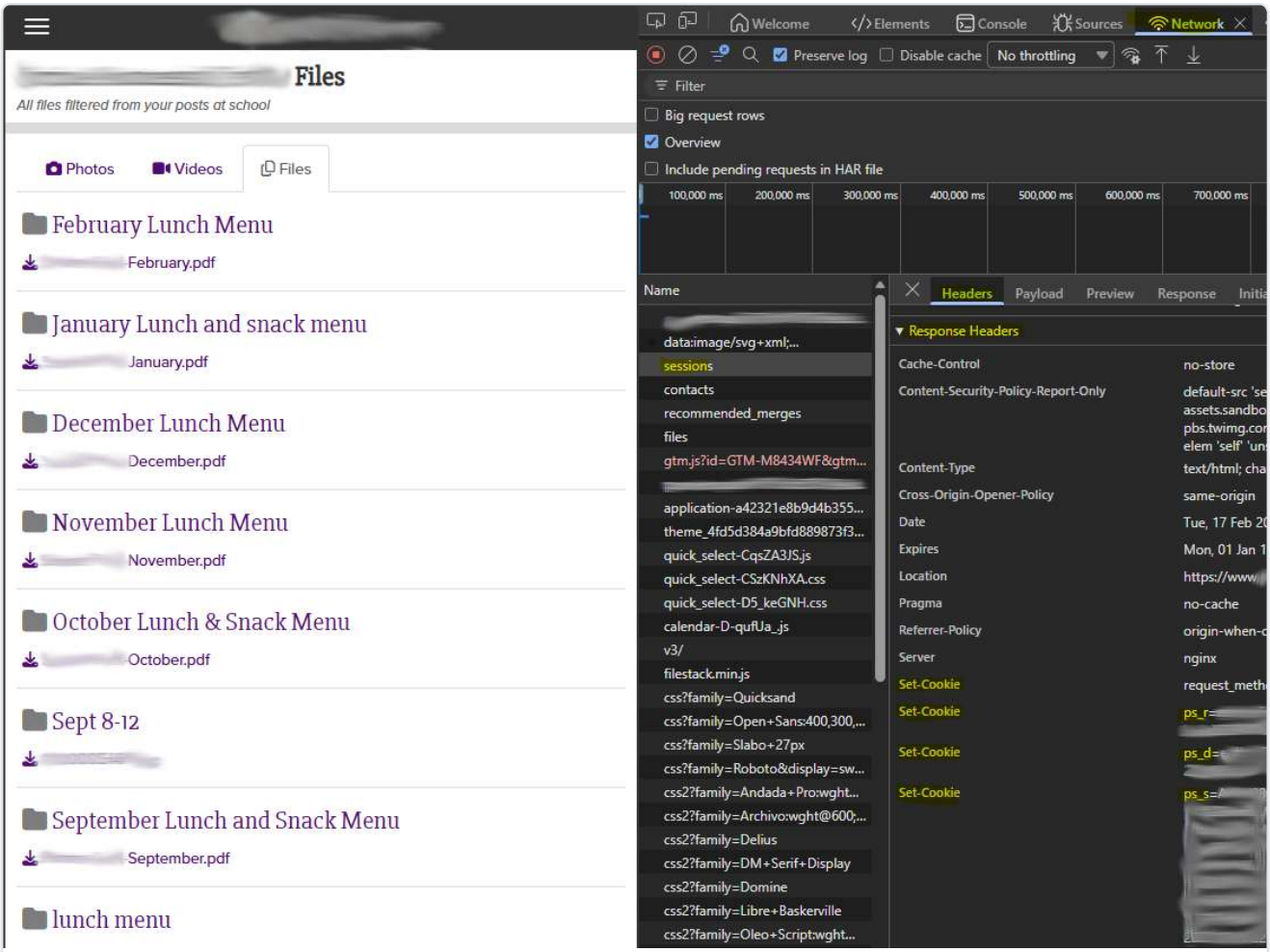
If you see...	It means...
Redirect to `/users/` or `/contacts/`	✅ Login SUCCESS
Redirect back to `/signin`	❌ Login FAILED - wrong credentials
Error message in HTML	❌ Login FAILED - invalid token or other error
Multiple `Set-Cookie` headers	✅ Authentication cookies being set

9 Extract Authentication Cookies

This is THE MOST IMPORTANT step! These cookies prove we're logged in and allow us to access protected pages.

Finding Cookies in the Response:

1. Still in the POST /sessions request details
2. Make sure you're on the **Headers** tab
3. Scroll to the **Response Headers** section
4. Look for ALL `Set-Cookie` entries



The response sets three cookies — ps_s, ps_r, and ps_d — all three are required for authenticated requests. Missing even one causes access denied errors

What We Found:

Set-Cookie Headers

```
Set-Cookie: ps_s=eWJhc2U2NGVuY29kZWRkYXRh...; Path=/; Expires=Mon, 15 Feb
Set-Cookie: ps_r=c29tZW90aGVyZGF0YQ...; Path=/; Expires=Mon, 15 Feb 2027;
Set-Cookie: ps_d=bW9yZWRhdGE...; Path=/; Expires=Mon, 15 Feb 2027; SameSit
Set-Cookie: request_method=; Path=/; Max-Age=0; Expires=Thu, 01 Jan 1970
```

Cookie Analysis:

- **ps_s:** Session cookie - the main authentication token
- **ps_r:** Remember cookie - helps maintain long sessions
- **ps_d:** Device cookie - tracks your device

- **request_method:** Being cleared (Max-Age=0) - this is normal
- **Expiration:** Valid for ~1 year - long-term authentication!
- **SameSite=Lax:** Security setting to prevent CSRF

🔴 **IMPORTANT:** We need ALL three cookies (ps_s, ps_r, ps_d) for subsequent authenticated requests! Missing even one cookie can cause access denied errors.

Cookie Format We Need to Construct:

Cookie Header Format

```
Cookie: ps_s=VALUE1; ps_r=VALUE2; ps_d=VALUE3
```

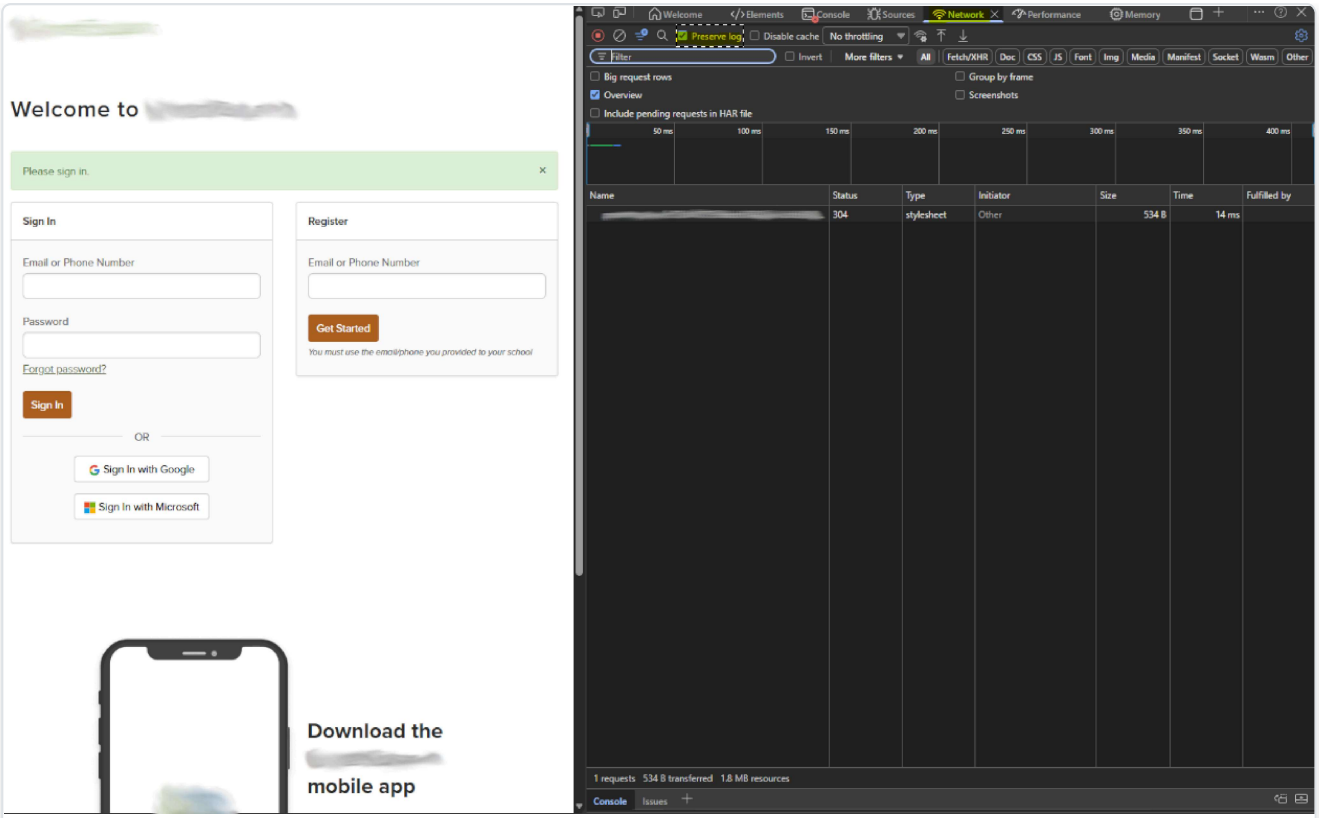
Notice: Just the name=value pairs, separated by semicolons and spaces. No "Path", "Expires", or other attributes!

10 Verify Cookies Work for Authenticated Requests

Let's confirm these cookies actually grant us access to protected pages.

Find the Next Request:

1. Look in the Network tab for the NEXT request after POST /sessions
2. It should be a GET request to a user-specific URL
3. Click on it to open the details



After login, the browser automatically follows the redirect to /users/... — a 200 OK here confirms authentication succeeded and cookies are working

Examine Its Request Headers:

Request Headers

```
GET https://your-school-portal.com/users/id/contacts HTTP/1.1
Host: your-school-portal.com
Cookie: ps_s=eWJhc2U2NGVuY29kZWRkYXRh...; ps_r=c29tZW90aGVyZGF0YQ...; ps_c
User-Agent: Mozilla/5.0...
```

> ✅ **SUCCESS:** Perfect! The browser automatically sent back all three authentication cookies. This proves the cookies work and grant access to protected pages.

Response Status Check:

Status Code	Meaning
200 OK	✅ Authenticated successfully - page loaded

302 or 30X	⚠️ Redirect - might be redirecting to login
401 Unauthorized	❌ Cookies invalid or missing
403 Forbidden	❌ Cookies valid but access denied

Part 3: Finding the PDF Download Links

11 Navigate to Menu Section

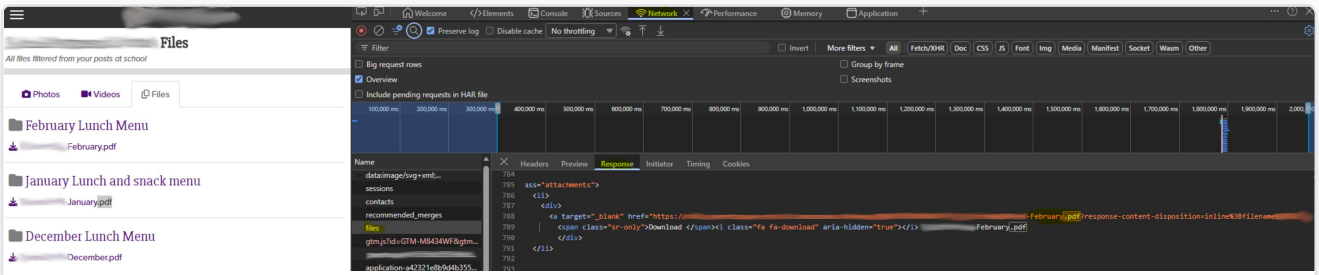
Now that we're authenticated, let's find where the lunch menu PDFs are located.

Preparation:

1. Stay in the Network tab
2. Make sure **Preserve log** is still enabled
3. Clear the log if it's getting cluttered (but keep preserve log on!)

Navigate:

1. Look for navigation items like "Files", "Documents", "Resources", "Lunch Menu"
2. Click on the menu/files section
3. Watch the Network tab for new requests



Navigating to the Files section of the portal while DevTools is open captures all the network requests — including the authenticated request to the menu page

9:10 AM 2/17/2026

Look For These Request Patterns:

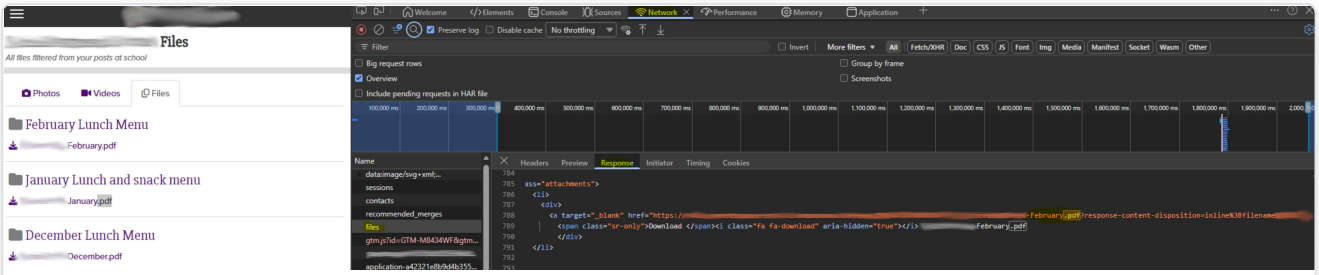
- URLs containing: `/feeds/` , `/files/` , `/documents/` , `/resources/`
- GET requests with Type: `document` or `html`
- Status: 200 OK

12 Inspect the Menu Page HTML

Once you've loaded the page with lunch menus, let's examine its HTML source.

Find the Page Request:

1. In Network tab, find the GET request to the menu page
2. Example: `GET /schools/id/feeds/files`
3. Click on it to open details
4. Click the **Response** tab



The Response tab shows the raw HTML returned by the server — this is the page our automation downloads and parses to find PDF links

You'll See:

The complete HTML source of the page, including all the links to PDF files!

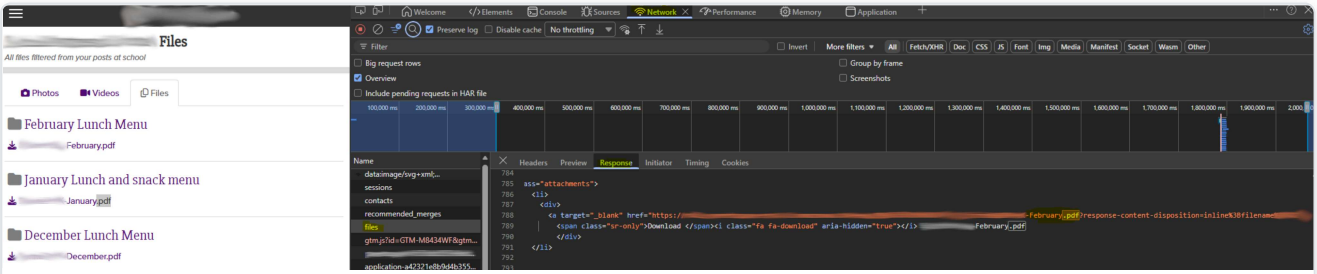
> 💡 **TIP:** If the HTML is minified (all on one line), you can copy it and use an online HTML formatter to make it readable. However, we'll search through it as-is.

13 Search for PDF Links

Let's find all PDF links in the HTML response.

Search Process:

1. In the Response tab, press `Ctrl+F`
2. Search for: `.pdf`
3. Press `Enter` or click the arrows to go through matches
4. Look at the HTML around each match



Searching for ".pdf" in the Response tab highlights every PDF link — the long S3 URLs with X-Amz- query parameters are visible here*

What We Discovered:

HTML

```
<a href="https://app-school-env.bucket.amazonaws.com/folder/February.pdf?r
   class="file-link"
   target="_blank">
  PreK-February.pdf
</a>
```

URL Analysis:

- **Host:** `app-school-env.bucket.amazonaws.com` - Hosted on Amazon S3
- **Path:** `/folder/February.pdf` - The actual file
- **Query Parameters:** Long string starting with `?response-content-disposition=...`
- **Filename Pattern:** Contains "February" - we can search for specific months!

Other PDFs Found:

- `-January.pdf`
- `March.pdf`
- `Grade 1 - February.pdf`
- `Grade 2 - February.pdf`

This pattern shows we can filter for our specific grade/class and month!

14 Understanding AWS Signed URLs

Those long query parameters aren't random - they're AWS S3 signed URL components.

Breaking Down the URL:

URL Components

```
https://app-school-env.bucket.amazonaws.com/folder/FILE.pdf
?response-content-disposition=inline%3Bfilename%3D%22FILE.pdf%22
&X-Amz-Expires=21600
&X-Amz-Date=20260215T192746Z
&X-Amz-Algorithm=AWS4-HMAC-SHA256
&X-Amz-Credential=token
&X-Amz-SignedHeaders=host
```

```
&X-Amz-Security-Token=token
&X-Amz-Signature=signature
```

What Each Parameter Means:

Parameter	Purpose	Value in Our Case
X-Amz-Expires	How long URL is valid	21600 seconds (6 hours)
X-Amz-Date	When URL was created	2026-02-15 at 19:27:46 UTC
X-Amz-Security-Token	Temporary AWS credentials	Long base64-encoded token
X-Amz-Signature	Cryptographic signature	Hex string proving authenticity

> ⚠️ **WARNING:** THE PROBLEM: When we tried downloading PDFs with these signed URL parameters, we got blank PDFs! The tokens were causing authentication conflicts with our session cookies.

The Discovery - Base URL Works Better:

We found that stripping everything after `.pdf` actually works!

Simplified URL

```
https://app-school-env.bucket.amazonaws.com/folder/February.pdf
```

> ✅ **SUCCESS:** When we use the base URL + our authentication cookies (from the login), we get the actual PDF file! The signed parameters were redundant and causing problems.

15 Test PDF Download Manually

Before automating, let's verify we can download the PDF manually.

Method 1: Click the Link

1. While still logged into the portal
2. Right-click on the PDF link
3. Select "Open link in new tab"
4. PDF should open in browser

> ✅ **SUCCESS:** If the PDF opens and shows the lunch menu, we know the link works and we're properly authenticated!

Method 2: Copy URL and Test Clean Version

1. Right-click the PDF link → "Copy link address"
2. Open a new tab and paste the URL
3. Manually remove everything after `.pdf` (all the query parameters)
4. Press Enter

Before

```
https://...February.pdf?response-content-disposition=...&X-Amz-Expires=...
```

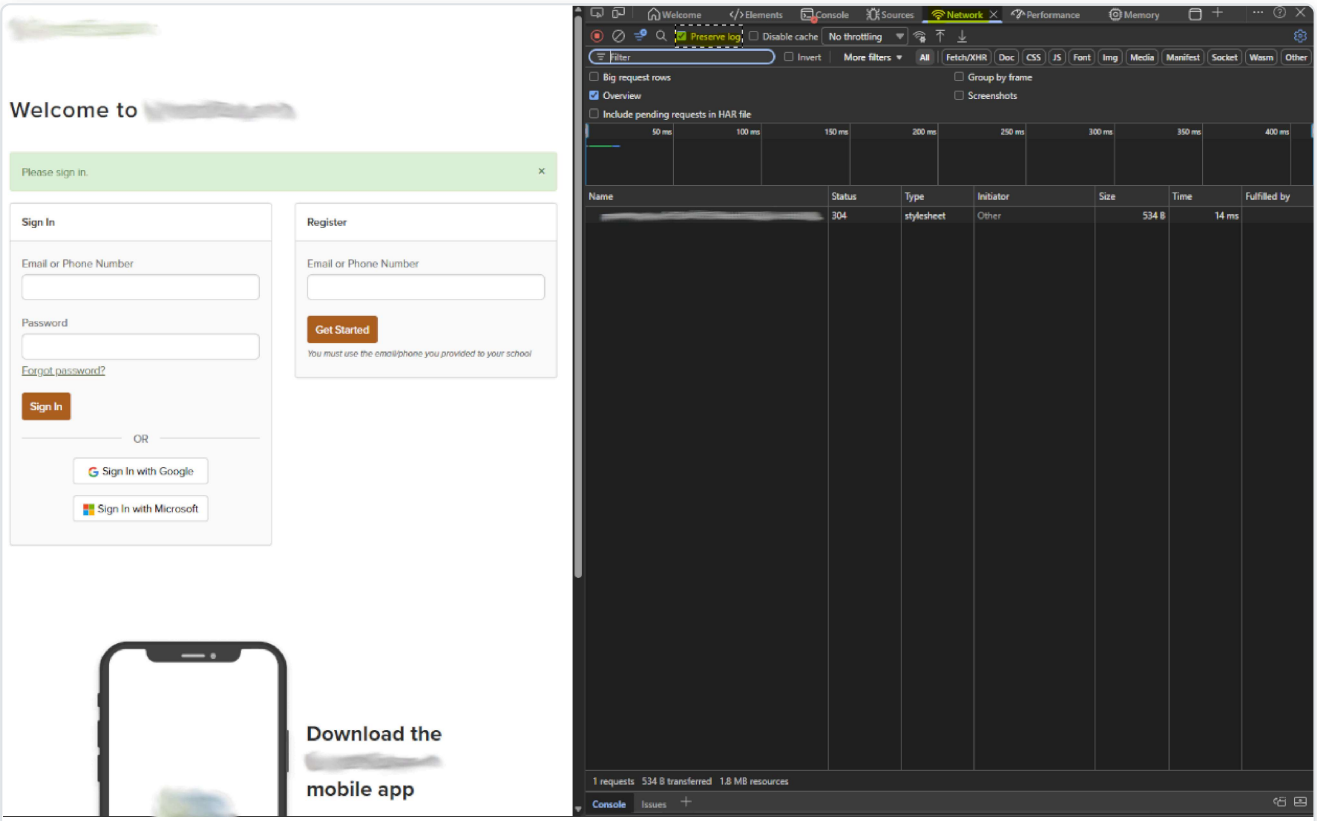
After

```
https://...February.pdf
```

> 💡 **TIP:** If the simplified URL works (PDF opens), you've confirmed that the signed URL parameters aren't necessary when you're already authenticated via cookies!

Method 3: Check Download in Network Tab

1. When PDF opens, look in Network tab
2. Find the GET request to the PDF URL
3. Check Response Status: should be 200 OK
4. Check Size: should match the PDF file size (e.g., 134 KB)
5. Click Response tab and verify it starts with `%PDF-1.7`



A 200 OK on the PDF request confirms the file downloaded correctly — checking that the response starts with %PDF-1.7 proves it's a real PDF and not an HTML error page

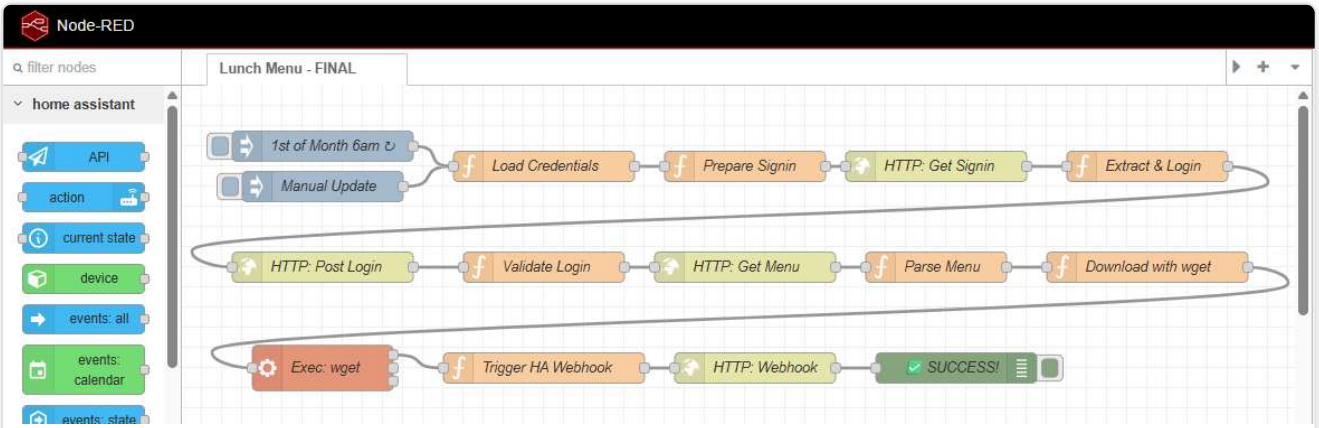
Part 4: Translating Findings to Node-RED

Now that we understand how the authentication works, let's convert our discoveries into a Node-RED flow.

16 Design the Flow Architecture

Flow Steps:

1. **GET /signin** → Fetch the login page
2. **Extract Token** → Parse HTML for authenticity_token
3. **POST /sessions** → Submit credentials with token
4. **Extract Cookies** → Get auth cookies from redirect
5. **GET /feeds/files** → Access menu page with cookies
6. **Parse HTML** → Find PDF URLs
7. **Clean URL** → Strip query parameters
8. **Download PDF** → Use wget with cookies



The complete Node-RED flow — left to right: credentials → signin → token extraction → login POST → cookie extraction → menu page → PDF parsing → wget download → Home Assistant webhook

17 Node-RED Implementation - Step by Step

Step 1: GET Signin Page

Function Node - "Prepare Signin"

```
msg.url = 'https://your-school-portal.com/signin';
msg.method = 'GET';
delete msg.headers;
delete msg.payload;
return msg;
```

Connect to: **HTTP Request** node configured for text response

Step 2: Extract CSRF Token

Function Node - "Extract Token"

```
const html = msg.payload;

// Use regex to find the authenticity token
const tokenMatch = html.match(/name="authenticity_token"[^>]*value="([^"]+

if (!tokenMatch || !tokenMatch[1]) {
    node.error('❌ CSRF token not found!');
    return null;
}

const token = tokenMatch[1];
node.warn('✅ Token extracted: ' + token.substring(0, 20) + '...');

msg.csrfToken = token;
return msg;
```

Step 3: Prepare Login POST

Function Node - "Prepare Login"

```
const username = '[email protected]';
const password = 'your-password';
const token = msg.csrfToken;

// Build form data exactly as browser sends it
const formData = new URLSearchParams();
formData.append('utf8', '✓');
formData.append('authenticity_token', token);
```

```
formData.append('session[email]', username);
formData.append('session[password]', password);
formData.append('commit', 'Sign In');

msg.url = 'https://your-school-portal.com/sessions';
msg.method = 'POST';
msg.payload = formData.toString();

// Critical headers
msg.headers = {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Origin': 'https://your-school-portal.com',
    'Referer': 'https://your-school-portal.com/signin',
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/5
};

return msg;
```

> 🔴 **IMPORTANT:** Store credentials securely! Use environment variables or Home Assistant secrets instead of hard-coding passwords.

Step 4: Extract Authentication Cookies

Function Node - "Extract Cookies"

```
// Check if login was successful
const responseUrl = msg.responseUrl || '';

if (!responseUrl.includes('/users/') && !responseUrl.includes('/contacts')
    node.error('❌ Login failed - redirected to: ' + responseUrl);
    return null;
}

node.warn('✅ Login successful!');

// Extract cookies from redirect
let allCookies = [];
if (msg.redirectList && msg.redirectList.length > 0) {
    const redirectCookies = msg.redirectList[0].cookies || {};

    for (const [key, val] of Object.entries(redirectCookies)) {
        const value = typeof val === 'object' ? val.value : val;
```

```
            allCookies.push(key + '=' + encodeURIComponent(value));
        }
    }

    const cookieHeader = allCookies.join('; ');
    node.warn('✅ Extracted cookies: ' + cookieHeader.substring(0, 50) + '...'

    // Save for later use
    msg.authCookies = cookieHeader;

    return msg;
```

Step 5: Access Menu Page

Function Node - "Get Menu Page"

```
    msg.url = 'https://your-school-portal.com/schools/YOUR_SCHOOL_ID/feeds/fil
    msg.method = 'GET';
    delete msg.payload;

    // Use authentication cookies
    msg.headers = {
        'Cookie': msg.authCookies,
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/5
    };

    return msg;
```

Step 6: Parse HTML for PDF URLs

Function Node - "Find PDF URLs"

```
    const html = msg.payload;

    // Find all PDF links
    const pdfPattern = /href=["'](https:\/\/.*?\.s3\.amazonaws\.com\/feeds\/[^
    const pdfs = [];
    let match;

    while ((match = pdfPattern.exec(html)) !== null) {
        pdfs.push(match[1]);
```

```
}

if (pdfs.length === 0) {
    node.error('❌ No PDFs found!');
    return null;
}

node.warn('✅ Found ' + pdfs.length + ' PDFs');

// Find specific month (February in this example)
const targetMonth = 'February';
let pdfUrl = null;

for (let i = 0; i < pdfs.length; i++) {
    if (pdfs[i].includes(targetMonth)) {
        pdfUrl = pdfs[i];
        break;
    }
}

if (!pdfUrl) {
    node.warn('⚠️ February PDF not found, using first PDF');
    pdfUrl = pdfs[0];
}

// CRITICAL: Strip query parameters
const cleanMatch = pdfUrl.match(/(https:\/\/[^?]+\.pdf)/);
if (cleanMatch) {
    pdfUrl = cleanMatch[1];
    node.warn('✂️ Stripped signed URL parameters');
}

node.warn('📥 Target PDF: ' + pdfUrl);
msg.pdfUrl = pdfUrl;

return msg;
```

Step 7: Download PDF with wget

Function Node - "Prepare Download"

```
const pdfUrl = msg.pdfUrl;
const cookies = msg.authCookies;
```

```
// Use wget via exec node - most reliable for binary files
msg.payload = `mkdir -p /share/lunch-menu && ` +
              `wget --header="Cookie: ${cookies}" ` +
              `--user-agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64) App
              `-O /share/lunch-menu/menu.pdf "${pdfUrl}" && ` +
              `ls -lh /share/lunch-menu/menu.pdf`;

return msg;
```

Connect to: **Exec** node (runs shell commands)

> 💡 **TIP:** Why wget instead of HTTP Request node? We discovered that Node-RED's HTTP Request node was corrupting binary PDF files. Using wget via exec node solved this completely!

Part 5: Common Pitfalls & Solutions

Pitfall 1: "Access Denied" or Blank PDFs

Symptoms:

- PDF downloads but file size is wrong (too small)
- PDF opens blank or shows error
- File contains HTML instead of PDF data

Root Cause:

Not sending authentication cookies with the download request, so S3 returns an "Access Denied" HTML page instead of the PDF.

How to Diagnose:

Check File Contents

```
head -c 200 /share/lunch-menu/menu.pdf
```

Should show: `%PDF-1.7` (actual PDF)

If shows: `<!DOCTYPE html>` (HTML error page)

Solution:

Correct wget Command

```
wget --header="Cookie: ps_s=...; ps_r=...; ps_d=..." \
     --user-agent="Mozilla/5.0..." \
     -O menu.pdf "https://s3-url.pdf"
```

Pitfall 2: "CSRF Token Invalid"

Symptoms:

- Login POST returns error
- Redirected back to /signin page
- Error message: "Invalid authenticity token"

Root Causes:

- Token not extracted correctly from HTML
- Token from old page load (expired)
- Token not properly included in POST data

Solutions:

1. **Verify extraction:** Add debug node after token extraction to see actual value
2. **Fresh token:** Always GET /signin immediately before POST /sessions
3. **Check regex:** Ensure regex pattern matches your portal's HTML

Debug Token

```
node.warn('Token: ' + msg.csrfToken);
// Should show long random string like: Ey_Gn3uE2ZBQ8EmyAFAMqP7vK9...
```

Pitfall 3: Cookies Not Being Captured

Symptoms:

- Login succeeds but next request fails
- "Not authenticated" errors on protected pages
- Redirected to login again

Root Cause:

Looking for cookies in wrong location. They're in `msg.redirectList[0].cookies`, NOT in `msg.responseCookies`!

Wrong:

❌ Incorrect Location

```
const cookies = msg.responseCookies;  // Usually empty!
```

Correct:

✅ Correct Location

```
const cookies = msg.redirectList[0].cookies;  // Contains auth cookies!
```

Debug:

Inspect redirectList

```
node.warn('redirectList: ' + JSON.stringify(msg.redirectList, null, 2));
```

Pitfall 4: PDF File Corruption

Symptoms:

- File size slightly wrong (e.g., 131KB instead of 137KB)
- PDF won't open or shows errors
- Binary data looks corrupted

Root Cause:

Node-RED's File node or HTTP Request node converting binary buffer to string, losing data in the process.

Solution:

Use `exec` node with `wget` command instead of HTTP Request + File nodes.

✅ **Reliable Method**

```
wget -O /path/to/file.pdf "https://url"
```

wget is designed for downloading files and handles binary data perfectly.

Pitfall 5: Signed URLs Causing Issues

Symptoms:

- Downloads work sometimes but fail randomly

- PDFs are blank even though authentication works
- Different results when testing manually vs automated

Root Cause:

AWS signed URL parameters (`X-Amz-*`) conflicting with session cookie authentication.

Solution:

Strip everything after `.pdf` in the URL:

JavaScript (in Function Node)

```
// Original URL with signed parameters
let url = "https://...file.pdf?response-content-disposition=...&X-Amz-Expi

// Extract just the base URL
const match = url.match(/(https:\/\/[^?]+\.pdf)/);
if (match) {
    url = match[1];  // Now: "https://...file.pdf"
}
```

Then use authentication cookies instead of signed URL for access.

Part 6: Verification & Testing

Testing Each Stage Independently

Test 1: Signin Page

Using curl

```
curl -v https://your-school-portal.com/signin
```

Look for:

- HTTP 200 OK status
- HTML response containing `authenticity_token`
- Set-Cookie header (initial session cookie)

Test 2: Token Extraction

Using grep

```
curl -s https://your-school-portal.com/signin | grep authenticity_token
```

Should output:

```
<input type="hidden" name="authenticity_token" value="...">
```

Test 3: Login POST

Using curl with form data

```
curl -v -X POST https://your-school-portal.com/sessions \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "utf8=✓&authenticity_token=TOKEN&session[email]=EMAIL&session[passwo
```

Look for:

- HTTP 302 redirect
- Location header pointing to /users/ or /contacts
- Multiple Set-Cookie headers (ps_s, ps_r, ps_d)

Test 4: Authenticated Request

Using curl with cookies

```
curl -v https://your-school-portal.com/schools/XXXXX/feeds/files \
  -H "Cookie: ps_s=...; ps_r=...; ps_d=..."
```

Look for:

- HTTP 200 OK
- HTML containing PDF links
- NOT redirected to /signin

Test 5: PDF Download

Using wget with cookies

```
wget --header="Cookie: ps_s=...; ps_r=...; ps_d=..." \
     -O test.pdf \
     "https://s3-url.pdf"
```

Verify result:

Check file type

```
file test.pdf
# Should say: "PDF document, version 1.7"

head -c 8 test.pdf
# Should show: %PDF-1.7
```

Appendix: Quick Reference

Browser DevTools Shortcuts

Action	Windows/Linux	Mac
Open DevTools	F12 or Ctrl+Shift+I	Cmd+Option+I
Network tab	Ctrl+Shift+E	Cmd+Option+E
Elements tab	Ctrl+Shift+C	Cmd+Option+C

Console tab	Ctrl+Shift+J	Cmd+Option+J
Search in page	Ctrl+F	Cmd+F
Clear network log	Ctrl+E	Cmd+E

Regular Expression Patterns

Purpose	Pattern
Extract CSRF token	`/name="authenticity_token"[^>]*value="([^"]+)"/`
Find PDF links	`/href=["'](https:\/\/.*?\.pdf[^"']*?)["']/gi`
Strip query parameters	`/(https:\/\/[^?]+\.pdf)/`
Find email in HTML	`/name="session\[email\]"/`

HTTP Status Codes

Code	Meaning	What to do
200 OK	Success	✅ Continue
302 Found	Redirect	✅ Check Location header
401 Unauthorized	Not authenticated	❌ Check cookies
403 Forbidden	Access denied	❌ Check permissions
404 Not Found	URL doesn't exist	❌ Check URL

Conclusion

Through careful inspection of browser network traffic and HTML source, we successfully reverse-engineered a complete authentication flow. The key discoveries were:

1. **Login Flow:** GET signin → Extract token → POST credentials → Extract cookies
2. **Form Fields:** `session[email]`, `session[password]`, `authenticity_token`
3. **Cookie Handling:** Multiple cookies from redirect response required for authentication
4. **PDF Location:** S3 URLs with signed parameters that needed to be stripped
5. **Download Solution:** Use wget with authentication cookies for reliable bina