Overcoming access control in web APIs

How to address security concerns using Sanic

Adam Hopkins
class Adam:

    def __init__(self):
        self.work = PacketFabric("Sr. Software Engineer")
        self.oss = Sanic("Core Maintainer")
        self.home = Israel("Negev")

    async def run(self, inputs: Union[Pretzels, Coffee]) -> None:
        while True:
            await self.work.do(inputs)
            await self.oss.do(inputs)

    def sleep(self):
        raise NotImplemented

- **PacketFabric** - Network-as-a-Service platform; private access to the cloud; secure connectivity between data centers


- **GitHub** - /ahopkins

- **Twitter** - @admhpkns
What we will **NOT** cover?

- TLS
- Password and other sensitive information storage
- Server security
- SQL injection
- Data validation
1. Authentication - Do I know who this person is?

2. Authorization - Should I let them in?
@app.get("/protected")
async def top_secret(request):
    return json({"foo":"bar"})
@app.get("/protected")
async def top_secret(request):
    return json({'foo': 'bar'})

curl localhost:8000/protected -i
HTTP/1.1 200 OK
Content-Length: 13
Content-Type: application/json
Connection: keep-alive
Keep-Alive: 5

{"foo":"bar"}
async def do_protection(request):
    ...

def protected(wrapped):
    def decorator(handler):
        async def decorated_function(request, *args, **kwargs):
            await do_protection(request)
            return await handler(request, *args, **kwargs)
        return decorated_function
    return decorator(wrapped)

@app.get="/protected"
@protected
async def top_secret(request):
    return json({"foo": "bar"})
async def do_protection(request):
    ...

@app.middleware('request')
async def global_authentication(request):
    await do_protection(request)
<table>
<thead>
<tr>
<th>Authentication</th>
<th>401</th>
<th>Unauthorized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authorization</td>
<td>403</td>
<td>Forbidden</td>
</tr>
</tbody>
</table>
Remember!

<table>
<thead>
<tr>
<th>Authentication</th>
<th>Status Code</th>
<th>Status Text</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>401</td>
<td>Unauthorized</td>
</tr>
<tr>
<td>Authorization</td>
<td>403</td>
<td>Forbidden</td>
</tr>
</tbody>
</table>

from sanic.exceptions import Forbidden, Unauthorized

async def do_protection(request):
    if not await is_authenticated(request):
        raise Unauthorized("Who are you?")

    if not await is_authorized(request):
        raise Forbidden("You are not allowed")
curl localhost:8000/protected -i
HTTP/1.1 401 Unauthorized
Content-Length: 49
Content-Type: application/json
Connection: keep-alive
Keep-Alive: 5

{"error":"Unauthorized","message":"Who are you?"}
async def is_authenticated(request):
    """How are we going to authenticate requests?"""
Common authentication strategies

- Basic
- Digest
- Bearer
- OAuth
- Session
Common authentication strategies

- Basic
- Digest
- Bearer
- OAuth
- Session
Forget what you know!

You didn't
See anything
Train pass

**Session based**
- Single Ride
- Point A to Point B

**Bearer Non-session based**
- All day pass
- Off and on at any stop
**Session based**

aka Single Ride

---

Client

/\login using credentials

Server

persist session details

Datastore

session_id

/\protected using session_id

confirm session_id

OK

protected resource

Client

Server

Datastore
Bearer Non-session based

/client

/login using credentials

generate token

token

/protected using token

confirm authenticity, etc

protected resource

Client

Server
Hold that thought ...
Let's decide on an auth strategy...

1. Who will **consume** the API?
   - Applications? Scripts? People?
2. Do you have **control** over the client?
3. Will this power a **web browser** frontend application?
What we really want to know is...

**Direct API v. Browser Based API** (or both)
**Direct API**

- Fewer security concerns
- Scripts, mobile apps, non-browser clients
- More technically sophisticated users
- API key or JWT

```
$ curl https://foo.bar/protected
```

**Solved**

**Browser Based API**

- More security concerns (CSRF, XSS)
- Web applications
- Lesser technically sophisticated users
- Session ID or JWT

```
fetch('https://foo.bar/protected').then(r => {
    console.log(response)
})
```

**Unsolved**
**Browser Based API Concerns**

1. How should the browser **store** the token? *(XSS)*
   - Cookie, localStorage, sessionStorage, in memory

2. How should the browser **send** the token? *(CSRF)*
   - Cookie, Authentication header
Typical recommendations

Session based

- **Stored:** Set-Cookie: token=\(<\text{TOKEN}\>\)
- **Sent:** Cookie: token=\(<\text{TOKEN}\>\)
- Subject to **CSRF**
- Fixed with: X-XSRF-TOKEN: <\text{CSRFTOKEN}>\)

Solved

Non-session based

- **Stored:** JS accessible
- **Sent:** Authorization: Bearer \(<\text{TOKEN}\>\)
- Subject to **XSS**

Unsolved
How do we authenticate?

- Session based v. Non-session based
- Direct API v. Browser Based API (or both)
- API key v. Session ID v. JWT
How do we authenticate?

- Session based v. Non-session based
- Direct API v. Browser Based API (or both)
- API key v. Session ID v. JWT

Solutions:

Direct API using API key in **Authorization** header
Browser Based API using session ID in cookies
How do we authenticate?

- Session based v. Non-session based
- Direct API v. Browser Based API (or both)
- API key v. Session ID v. JWT

Solutions:

Direct API using API key in Authorization header
Browser Based API using session ID in cookies

But what about:

- Both Direct API and Browser Based API?
How do we authenticate?

- Session based v. Non-session based
- Direct API v. Browser Based API (or both)
- API key v. Session ID v. JWT

Solutions:

- Direct API using API key in `Authorization` header
- Browser Based API using session ID in cookies

But what about:

- Both Direct API and Browser Based API?
- Browser Based API using non-session tokens, aka JWTs?
WHAT IF I TOLD YOU WE CAN SECURE

JWT FROM BOTH CSRF AND XSS?
Anatomy of a JWT

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
Anatomy of a JWT

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

{
    "alg": "HS256",
    "typ": "JWT"
}

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ

{
    "sub": "1234567890",
    "name": "John Doe",
    "iat": 1516239022
}

SflKxwRJSMeKKF2QT4fwMeJf36POk6yJV_adQssw5c

signature
Anatomy of a JWT

Set-Cookie access_token=
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ; Secure

Set-Cookie access_token_signature=
SflKxwRJSMeKGF2QT4fwpMeJf36POk6yJV_adQssw5c; Secure;HttpOnly
Anatomy of a JWT

Set-Cookie access_token=
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ; Secure

Set-Cookie access_token_signature=
SflKxwRJSMeKkF2QT4fwpMeJf36POk6yJvV_adQssw5c; Secure; HttpOnly

TELL ME MORE ABOUT THESE COOKIES
Split JWT cookies

header_payload, signature = access_token.rsplit(".", maxsplit=1)

set_cookie(
    response, "access_token", header_payload, httponly=False
)

set_cookie(
    response, "access_token_signature", signature, httponly=True,
)

set_cookie(
    response, "csrf_token", generate_csrf_token(), httponly=False,
)  # Do we even need this? Perhaps not!

def set_cookie(response, key, value, config, httponly=None):
    response.cookies[key] = value
    response.cookies[key]["httponly"] = httponly
    response.cookies[key]["path"] = "/
    response.cookies[key]["domain"] = "foo.bar"
    response.cookies[key]["expires"] = datetime(...)  
    response.cookies[key]["secure"] = True
We found a winner

Non-session Stateless JWT based

- **Stored**: JS-accessible 2 cookies
- **Sent**: Authorization: Bearer <TOKEN> 2 cookies
  Also, 1 token via Header for CSRF protection
- **Subject to**: Secured from **XSS**

Solved
def extract_token(request):
    access_token = request.cookies.get("access_token")
    access_token_signature = request.cookies.get("access_token_signature")

    return f"{access_token}.{access_token_signature}"

def is_authenticated(request):
    token = extract_token(request)

    try:
        jwt.decode(token, ...)
    except Exception:
        return False
    else:
        return True
def do_protection(request):
    if not is_authenticated(request):
        raise Unauthorized("Who are you?")

    if not isAuthorized(request):
        raise Forbidden("You are not allowed")

    if not is_pass_csrf(request):
        raise Forbidden("You CSRF thief!")
def is_authorized(request):
    """How shall we do this?"""
Structured Scopes

user:read:write

namespace:action(s)
Structured Scopes

**user**: read: write

namespace: action(s)

**user**: read
Structured Scopes

user:read:write

namespace:action(s)

user:read

Pass
from sscopes import validate

is_valid = validate("user:read:write", "user:read")
print(is_valid)
# True
def is_authorized(request, base_scope):
    if base_scope:
        token = extract_token(request)
        payload = token.decode(token, ...)

        return validate(base_scope, payload.get("scopes"))

    return True
@app.get("/protected")
@protected("user:read")
async def top_secret(request):
    return json({'foo': 'bar'})
@app.get("/protected")
@protected("user:read")
async def top_secret(request):
    return json({'foo': 'bar'})

fetch('https://foo.bar/protected').then(async response => {
    console.log(await response.json())
})
There must be a better way
There must be a better way

pip install sanic-jwt
from sanic_jwt import Initialize, decorators

async def authenticate(request):
    """Check that username and password are valid"""

async def retrieve_user(request):
    """Get a user object from DB storage"""

async def my_scope_extender(user):
    return user.scopes

app = Sanic()
Initialize(
    app,
    authenticate=authenticate,  # sanic-jwt required handler
    retrieve_user=retrieve_user,
    add_scopes_to_payload=my_scope_extender,
    cookie_set=True,  # Set and accept JWTs in cookies
    cookie_split=True,  # Expect split JWT cookies
    cookie_strict=False,  # Allow fallback to Authorization header
)

@app.get("/protected")
@decorators.scoped("user:read")
async def top_secret(request):
    ...

<table>
<thead>
<tr>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://foo.bar/auth">https://foo.bar/auth</a></td>
<td># Login with username/password</td>
</tr>
<tr>
<td><a href="https://foo.bar/auth/verify">https://foo.bar/auth/verify</a></td>
<td># Verify a valid JWT was passed</td>
</tr>
<tr>
<td><a href="https://foo.bar/auth/me">https://foo.bar/auth/me</a></td>
<td># View details of current user</td>
</tr>
<tr>
<td><a href="https://foo.bar/protected">https://foo.bar/protected</a></td>
<td># Must have user:read access</td>
</tr>
</tbody>
</table>