Web Application Security 101

Ben Eldritch

Whoami /all

About Me:

- Not from Virginia but loving it!
- Currently work at Raytheon Technologies (RTX)
- Bushcrafter by day, hacker by night

Stuff I've Done:

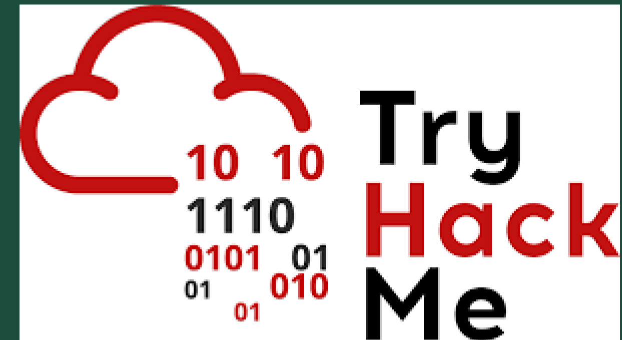
- GICSP, OSCP, Pentest+, CySA+, Sec+, Network+, etc…
- Roanoke Bsides 2023!
- Iowa State Cyber Defense Competitions
- RISE, NCL, Sans, Google CTFs
- Webapp pentesting/Bug bounties

Kontrabear, BenTheCyberOne

Thissiteissafe.com

Agenda

1. OWASP Overview
2. Secure Coding
3. Secure Design
4. Negative Outcomes
5. DEMO: Gilgamesh
6. Proper JWT Usage
7. Securing GraphQL Endpoints
8. Proper Validation



Gilgamesh Demo Room:

https://tryhackme.com/jr/gilgamesh

OWASP

- Open Worldwide Application Security Project

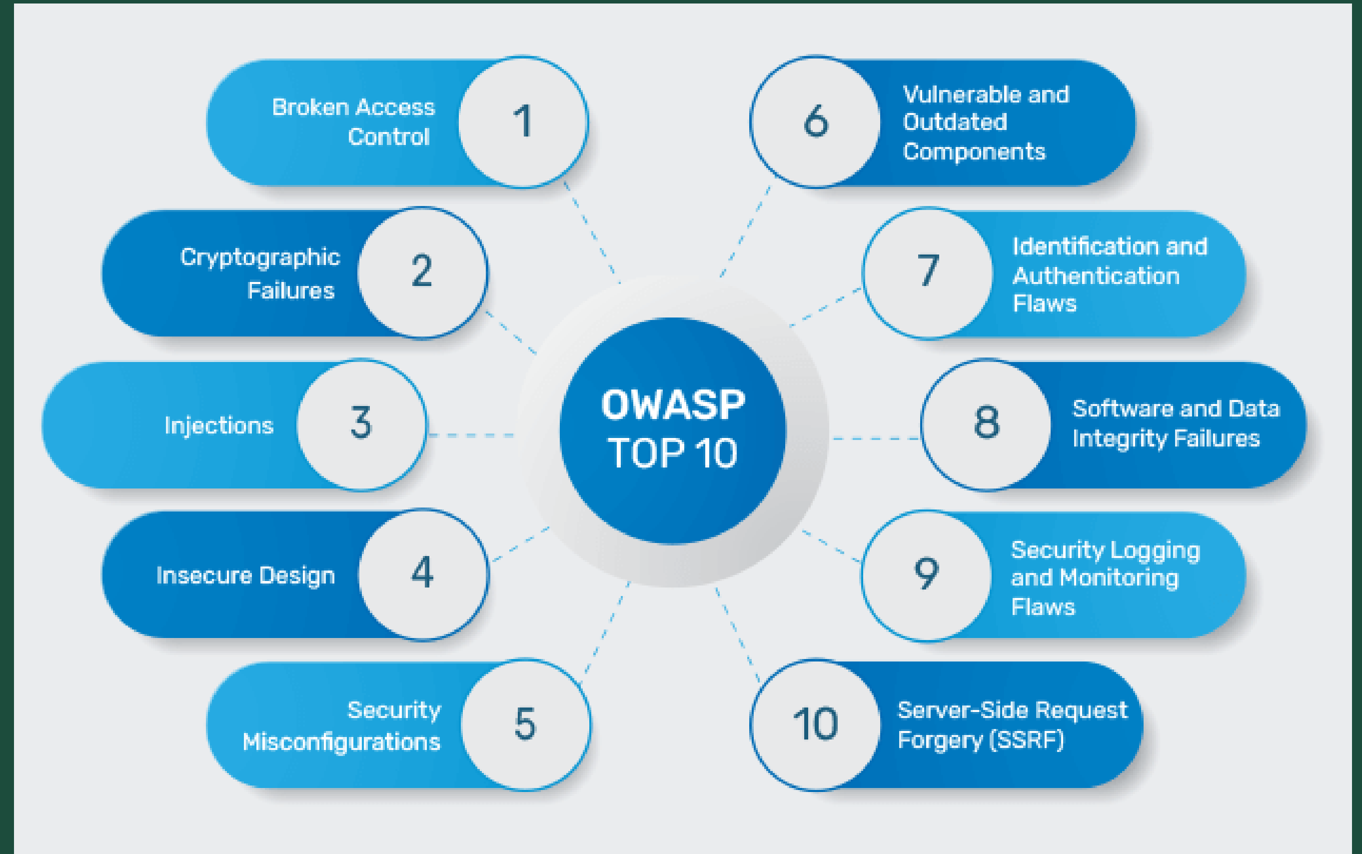
- “The global open community that powers secure software through education, tools, and collaboration”

- Foundation working to improve the security of software through open-source tools, workshops, providing best practices, etc.

- Releases a new (in)famous “OWASP Top 10” list every 3-4 years

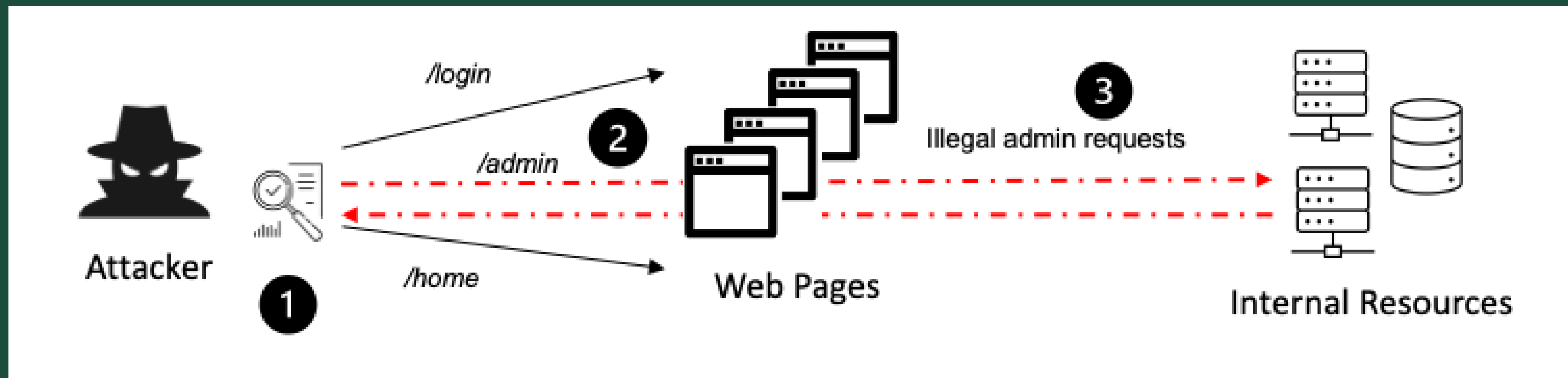
OWASP Top 10

- List of the top 10 most critical security risks to applications (often the most found as well)
- One of the best documents to adopt for web application security and secure design
- Most security scanning tools base their scanning profiles off of the OWASP Top 10
- Distinct categories based on each security risk, though some may appear to overlap



Our Top 4

A01 Broken Access Control
A02 Cryptographic Failures
A03 Injection
A04 Insecure Design



Secure Coding

- Projects should avoid known vulnerable and unknown libraries
  - https://www.fortinet.com/blog/threat-research/malicious-packages-hiddin-in-npm
  - Outdated packages can cause severe damage!

- Ensuring sensitive data is encrypted during transit and rest, all input is sanitized and validated, etc

- Utilizing Static Application Security Testing (SAST) tools on newly developed source code/projects
  - CodeSonar, Coverity, Gitlab
  - Something is better than nothing!

```
"name": "@zola-helpers/client",
"version": "1.0.1",
"description": "xxx haii",
"main": "index.js",
"type": "module",
"scripts": {
  "postinstall": "node index.mjs"
},
"keywords": [

],
"author": "",
"license": "ISC",
"dependencies": {
  "node-fetch": "^3.3.1"
}
```

```
{
  "name": "@expue/core",
  "version": "0.0.3-alpha.0",
  "description": "Latest v0.0.3-alpha.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "preinstall": "node index.js"
  },
  "author": "Anonymous",
  "license": "ISC"
}
```

```
const express = require('express');
const bodyParser = require('body-parser');
const mongoSanitize = require('express-mongo-sanitize');

const app = express();

app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());


app.use(mongoSanitize());
```

Secure Design

- Sure, your code may be solid. But is the logic?
- This can be very time consuming, but well worth it
- Most insecure logic/design issues will not be found on security scanners

```
app.post("/error",(req, res) => {
  const filename = req.body.filename;
  fs.readFile(filename,"utf8", (err, data) =>{
    if(err){
      console.error(err);
      return res.status(404).send("No valid error logfile found for this host.");
    }
    return res.render('error.ejs',{file: data});
  })
})
```

Perfect example of bad code and bad design!

Secure Design

- Isolate users wherever possible
  - RATE LIMIT
- Threat model each component in the application
  - If a user somehow gets into this, what can they do? What are my controls to alert me or stop them?
- Thoroughly design security rules, checks and access controls to each route/API gateway
- Ensure all publicly accessible routes are behind proper security gateways/Web Application Firewalls (WAFs)

Example 1 - just the code (no localization in the message):

The verification code is: **918135**

Example 2 - use the code in a clickable link to trigger a verification check:

Verify Email Now

…a few minutes later…

```
Get /api/user/verifyEmail?email=example@test.com HTTP/1.1
Host: reallycool.host:8080
Content-Length: 92
Accept: application/json, text/plain, */*
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/110.0.5481.178 Safari/537.36
Content-Type: application/json
Origin: http://reallycool.host:8080
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

{"verification_code":"§123456§"}
```

Brute forcing the code…

```
HTTP/2 200 HTTP/2
Content-Type: application/json; charset=UTF-8
Server: Apache
Vary: Accept-Encoding
Connection: close
Content-Length: 15

{"success": true}
```

Negative Outcomes

- Webapps can be one of the most silent ways for attackers to gain a foothold
- Webapps can be modified to do damage not just internally, but to external facing users as well
- Webapps typically run in a stack – where malicious users could pivot/gain more privilege
    - Connected cloud solutions
    - Databases
    - Source Code/Cryptographic Keys

GILGAMESH

"Gilgamesh is a brand new cloud service for OT equipment. With Gilgamesh, we promise to keep your devices alive no matter the cost! Simply connect your OT devices in your factory to the Gilgamesh Cloud and presto - factory management at the convenience of your home! We wondered why this kind of thing doesn't exist in today's market, so we figured we'll be the pioneers of this new age of connected factories! Gilgamesh: Keeping your factories ~~safe~~ up....forever!"

tryhackme.com/jr/gilgamesh

Proper JWT Usage

- JSON Web Tokens (JWT) are often used as access tokens and ID tokens

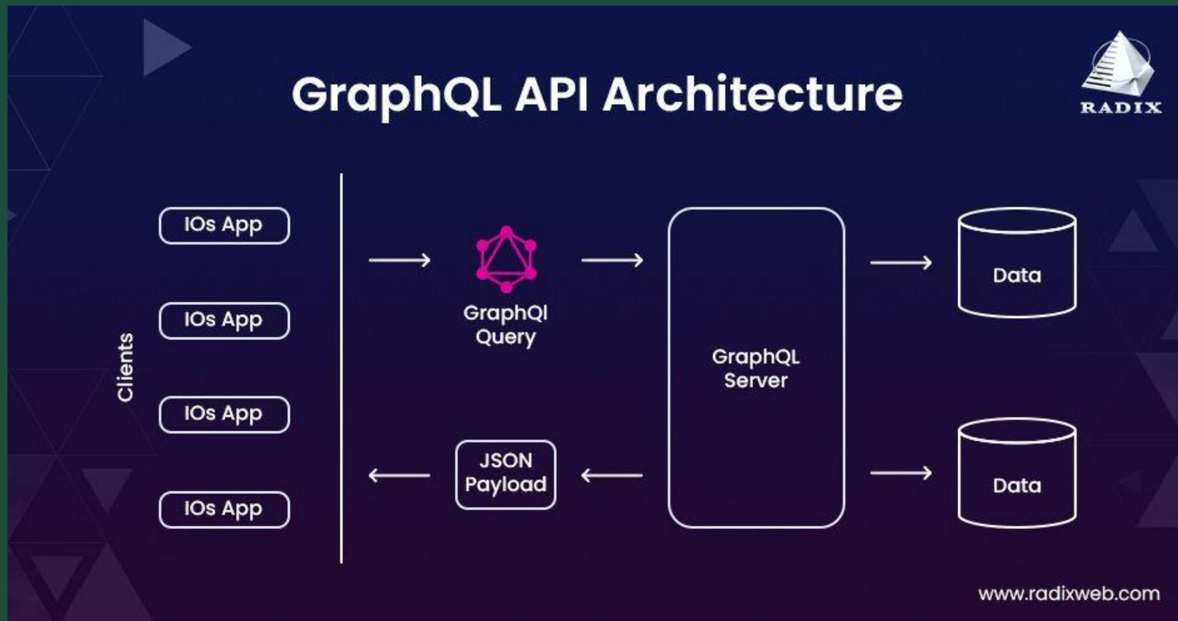
- They can be encrypted into JSON Web Encryption (JWE) type tokens, however JSON Web Signature (JWS) type tokens are often the go-to
  - JWE encrypts tokens with asymmetric encryption keys and cannot be decoded without them, whereas JWS utilizes digital signatures via HMAC or ECDSA/RSA (SHA-256/512) and can be decoded into plaintext with ease

```
[header].[payload].[signature]
```

  - JWS type tokens should really be signed with ECDSA/RSA key-pairs
    - Private key is utilized to sign tokens, public key is used to validate
    - RSA is much faster – important to web developers
    - Can be a bump in the road with fast scaling systems

  - If utilizing HMAC, ensure the signing key is:
    - A long, randomly generated multi-character string
    - Rotated frequently

Securing a GraphQL Endpoint



- The big 2 A's: Authentication and Authorization
  - Ensure only authenticated users can query the GraphQL endpoint
  - Check if the user *should* be allowed to perform a certain query
- Disable the playground/landing page for all production/external facing endpoints (unless it is a supplied service)
- DISABLE INTROSPECTION
- Mask any potential errors in queries (formatError API)
- Monitor for any anomalous behavior

Even with Introspection disabled…

```
{
  "operationName":"Identity",
  "variables":{
    "input":{
      "wid":"[redacted]",
      "mpid":"1"
    }
  },
  "query":
  "query Identity($input: ProfileInput) {\n  profile(input: $input) {\n    ... on Profile {\n      availableIdentities {\n        ... on BusinessIdentity {\n          isDenylisted\n          isSuspended\n          type\n          avatar {\n            url\n            __typename\n          }\n          displayName\n          handle\n          id\n          profileBackgroundPicture\n          balance {\n            userBalance {\n
```

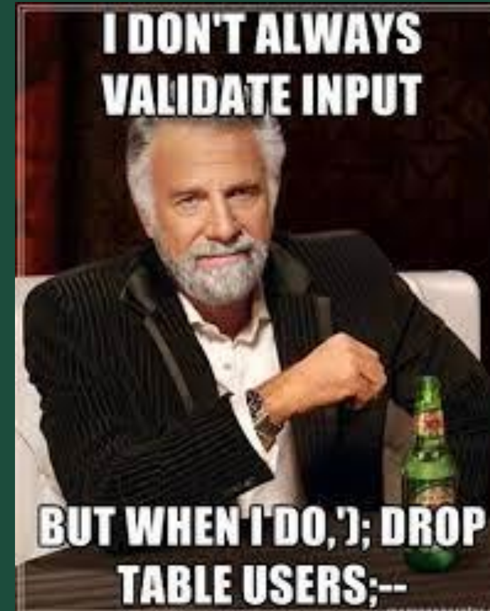
```
16
17 {
  "errors":[
    {
      "message":
      "Variable \"$input\" got invalid value { wid: \"[redacted]\", mpid: \"1\" }; Field \"wid\" is not defined by type \"ProfileInput\". Did you mean \"id\"?",
      "locations":[
        {
          "line":1,
          "column":16
        }
      ],
      "extensions":{
        "code":"BAD_USER_INPUT"
      }
```

Why yes! I did mean “id”!
My bad ☺

You’ll have to patch this behavior yourself ☺

Proper Validation

- Rule #1: Never trust anybody
  - If user data comes in, always clean it
  - Validation is better than sanitization
- Ensure validation middleware/functions are placed in every route user-fed data comes in
  - Don’t forget about the cookies!
- Enable your local “strict mode”
- Always validate input on the **server-side**
- Escape dangerous characters in the output




```
const express = require('express');
const { check } = require('express-validator');
const app = express();

app.use(express.json())

app.post('/form', [
  check('name').isLength({ min: 3 }).trim().escape(),
  check('email').isEmail().normalizeEmail(),
  check('age').isNumeric().trim().escape()
], (req, res) => {
  const name  = req.body.name
  const email = req.body.email
  const age   = req.body.age
})
```

Questions?

Thank you!

walkthrough:

thissiteissafe.com/gilgamesh-walkthrough/

Ben Eldritch

BenTheCyberOne/KontraBear on Roanoke Discord!

thissiteissafe.com