Writing secure JavaScript

Deep dive into vulnerabilities

Andrii Romasiun

About me

- Fullstack JavaScript Engineer
- Open-source countributor

/in/andriir

@blaumaus_

@blaumaus

Writing secure JavaScript

- JavaScript is the most **widespread** programming language in the world. It is crucial to know how to use this language **securely**.
- JavaScript has the most developed ecosystem, with more than **3 million** public packages and libraries.
- JavaScript is the fundamental technology for writing web applications and is used for writing APIs, desktop and mobile applications.

Why security in JS is important

- **30,000** websites get hacked **daily**.
- The majority of the attacks are executed by bots and can be **easily prevented** with the right knowledge.

XSS - what is it?

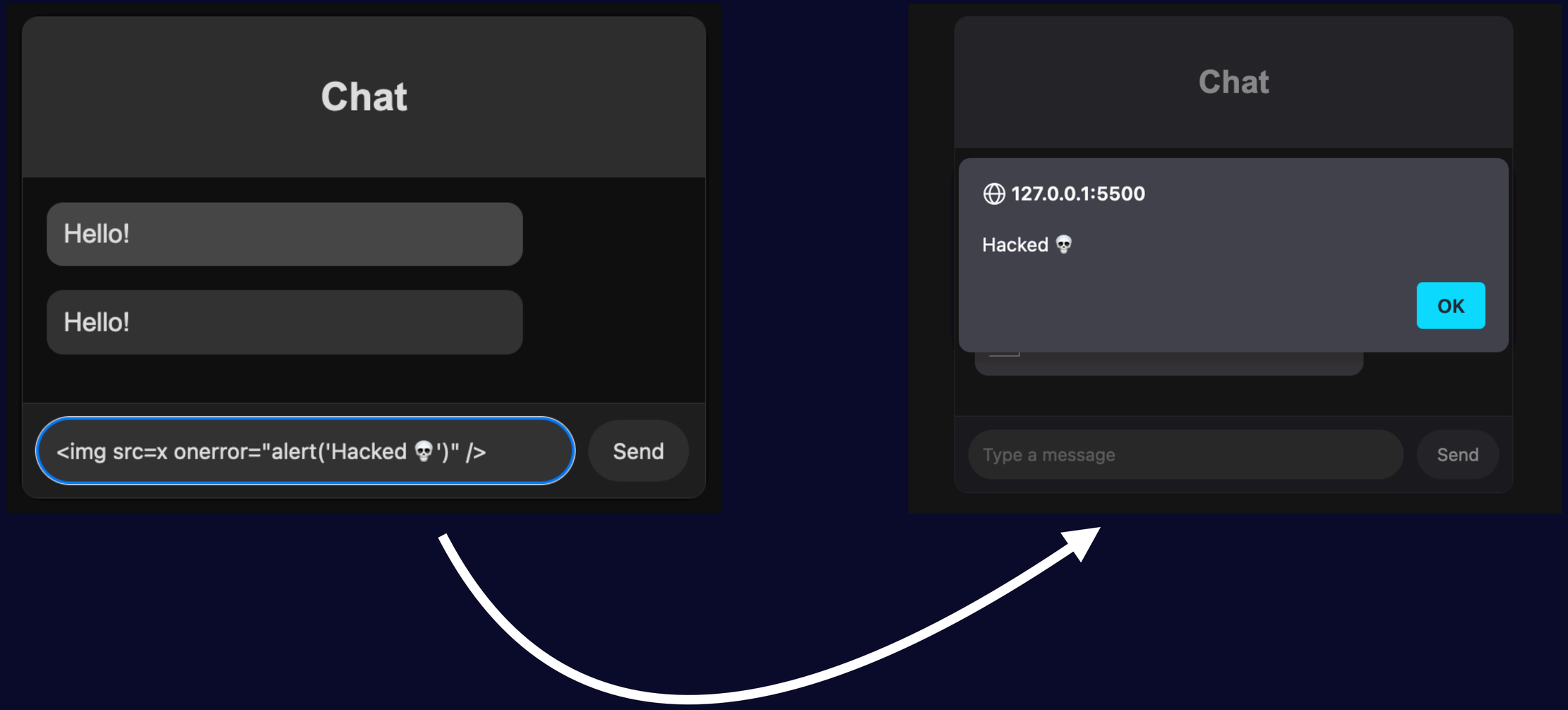
Cross-Site Scripting

- XSS (Cross-Site Scripting) is one of the most common vulnerabilities in client-side JavaScript. XSS attacks are type of injections, in which attacker can add custom JS code to legitimate web applications.
- By successfully exploiting XSS, the attacker can steal victims cookies or session information, redirect them to any other site or do anything else.

Kinds of XSS

- Stored XSS - user sends a malicious JS, it's stored in the DB (e.g. chat) and displayed to other people.
- Reflected XSS - commonly stored in the URL as a query parameter.
- DOM-based XSS - stored as a URI fragment (#). Impossible to detect server-side because fragments are never sent to the server.

XSS - example


Chat
Hello!
Hello!

Send
Chat
127.0.0.1:5500
Hacked 💀
OK
Type a message
Send

XSS - Real life cases

- In February 2019 an XSS was found in Google Search. It was reported to Google and immediately fixed by them a few days later.
- This XSS allowed to execute any custom JavaScript code, and had it been found by a potential attacker, the outcome could be devastating.

XSS - How to prevent?

- Always sanitise user content. Make sure to encode characters like **<**, **>**, **&**, etc. with their HTML unicode representation, like **`&#60;`** or **`&#62;`**
- Implement Content-Security-Policy to make sure that browser executes only JS code that you want. For example, setting CSP to `Content-Security-Policy: script-src 'self' https://www.jsdelivr.com/` will prevent any inline JS from being executed.
- Always treat user's input as text, not HTML. For example, use JavaScript's `.textContent` property instead of `.innerHTML` (in React, it should be `<div>{content}</div>` instead of `<div dangerouslySetInnerHTML={ __html: content } />`

Mass Assignment - What is it?

- Many frameworks and ORMs automate the process of updating records in the database. For example, you can simply pass an object and it will overwrite the row in the database.
- This functionality opens up a door that allows attacker to update fields they should not be allowed to update, which can let them escalate their privileges, grant paid features of your website to their account or lots of other things.

Mass Assignment - Example

```
@Post()
async updateUsername(
  @CurrentUserId() id: string,
  @Body() body: UsernameDTO,
): Promise<void> {
  await this.userService.update(id, body)
}
```

Such code will work fine, but it contains a vulnerability.

But assuming an application uses a field like `isAdmin` in the user table, an attacker could set it to `true` in the `body` object and administrative privilege would be granted.

Mass Assignment - Real life case

A mass assignment vulnerability was discovered in Uber in 2016. In the form where drivers could edit their profile details, like avatar image, an attacker could submit text fields like first_name, last_name, etc. and they would get applied without any verification from Uber's side.

Mass Assignment - How to prevent?

```
@Post()
async updateUsername(
  @CurrentUserId() id: string,
  @Body() body: UsernameDTO,
): Promise<void> {
  await this.userService.update(id, {
    username: body.username,
  })
}
```

Avoid using functions that bind user input into variables or internal objects automatically.

As a developer, you should explicitly use values from user input that should be updated and discard everything else.

Additionally, validation can be implemented to make sure that user does not supply fields they are not supposed to.

Path Traversal - What is it?

- On web servers, some files should be accessible and some should not. For example, many backend frameworks use a `public` folder for public assets such as styles, images or scripts.
- However, an attacker could manipulate the path to the file using “dot-dot-slash” notation, for example, by passing `/load-file?filename=../.env` as the file to download.

Path Traversal - Example

```
import * as express from 'express'
import * as fs from 'fs'
import * as path from 'path'

const app = express()

app.get('/load-file', (req, res) => {
    const filePath = req.query.filename
    const fullPath = path.join(__dirname, filePath)

    fs.readFile(fullPath, 'utf8', (err, data) => {
        if (err) {
            return res.status(500).send('Error reading file')
        }
        res.send(data)
    })
})
```

This code is vulnerable because it directly uses the input from the user to construct the file path without any validation or sanitisation.

An attacker could exploit this by providing a path like `../../../../etc/passwd` to access sensitive files outside the intended directory.

Path Traversal - Real life case

In 2018, a path traversal vulnerability was discovered in the Algolia self-destructing messaging service (msg.algolia.com). It allowed the attacker to read any file on the operating system, including those such as `/etc/passwd`.

This could be done by sending a GET request like `/static/..%252f..%252f..%252f..%252fetc/passwd` to their domain.

Path Traversal - How to prevent (1/2)?

- Use a content management system or a CDN instead of storing files and serving them from your server.
- If you need to store files on your server, each time the file is uploaded, create a random name for it and map it to the actual file path. Each time the file is requested, find the file path from the file name (e.g. in your database) and serve it. This would eliminate the need for absolute file paths.
- Use the principle of least privilege: create a new user in your operating system that only has access to predefined directories and files (such as the Assets directory). If a path traversal vulnerability is discovered, the impact would not be as drastic.

Path Traversal - How to prevent (2/2)?

```
import * as express from 'express'
import * as fs from 'fs'
import * as path from 'path'

const app = express()

const baseDirectory = path.join(__dirname, 'files')

app.get('/load-file', (req, res) => {
    const filePath = req.query.filename
    const fullPath = path.join(__dirname, filePath)

    // Ensure the path is within the base directory
    const resolvedPath = path.resolve(fullPath);
    if (!resolvedPath.startsWith(baseDirectory)) {
        return res.status(400).send('Invalid file path');
    }

    fs.readFile(fullPath, 'utf8', (err, data) => {
        if (err) {
            return res.status(500).send('Error reading file')
        }
        res.send(data)
    })
})
```

✅ Do this instead.

In your code, make sure that the file path is resolved within the intended directory.

A common method is to resolve the file path and check that it stays within a particular directory.

CSRF - What is it?

Cross-Site Request Forgery

- CSRF is a type of attack that occurs when users are tricked into performing actions on a third-party site that makes malicious requests to your site. All of your site's servers will see a request from an authenticated user, but the attacker is in control of what data is sent to your server on their behalf.
- If your server is not designed to be secure against CSRF attacks, it may result in users performing unwanted actions such as transferring funds, changing email addresses, etc.

CSRF - Example

```
app.post('/transfer', (req, res) => {
    const { account } = req.user || {}
    const { amount, toAccount } = req.body

    if (req.user) {
        transferFunds(account, toAccount, amount)
        res.send('Transfer successful')
    } else {
        res.status(401).send('Unauthorised')
    }
})
```

This code is vulnerable because it does not verify the source of the request.

An attacker could create a malicious web page with a form that submits a request to `/transfer` on behalf of the authenticated user.

CSRF - Real life case

In 2020, a critical CSRF vulnerability was found on the Glassdoor websites. It allowed an attacker to perform CSRF attacks on all actions on both job seeker and employer accounts.

An insecure CSRF token could be obtained from the server and all kinds of actions could be performed on behalf of Glassdoor users, such as leaving a review/salary information, editing a profile, etc.

CSRF - How to prevent?

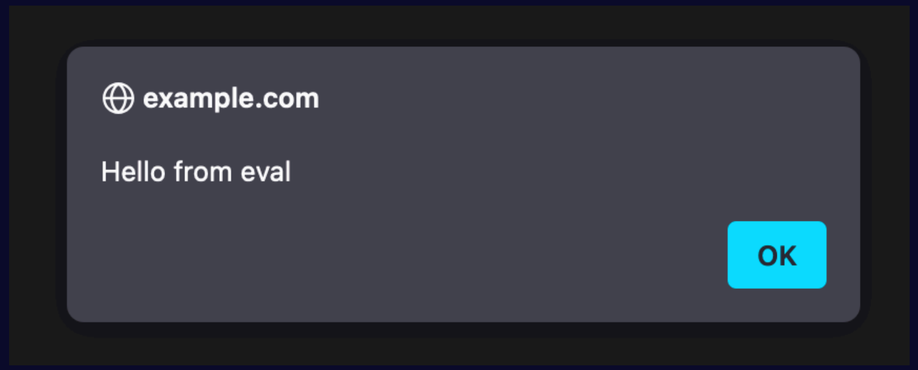
- Ensure that GET requests to your server are side-effect free (return information and don't change anything). This will prevent an attacker from phishing your users into a maliciously crafted URL.
- Set up anti-CSRF tokens for your endpoints: these are unique tokens for each request that are validated by the server. This ensures that the request comes from the authenticated user.
- Set cookies with the `SameSite` attribute to `Strict` or `Lax`. This mitigates CSRF by ensuring cookies are not sent with cross-site requests.

Remote Code Execution - What is it (1/2)?

Human-readable code exists as text before it's executed. Some programming languages compile the code into binary before executing it, while others interpret it line by line as it is. Some languages even allow some code to be executed as a string (for example, the `eval` function in JS).

```
const text = 'alert("Hello from eval")'

eval(text)
```



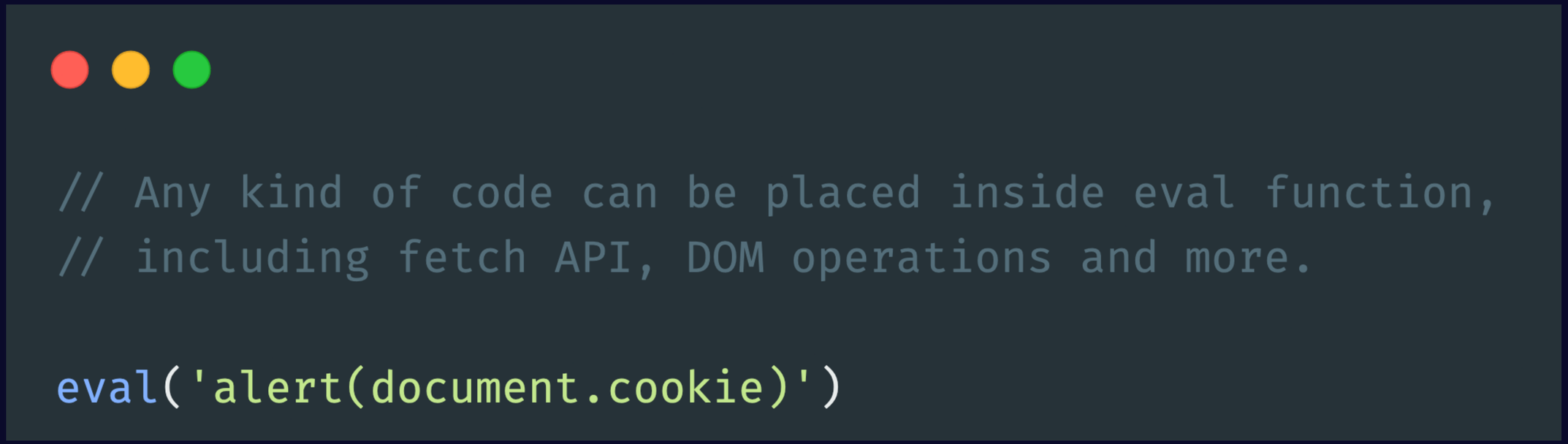
Remote Code Execution - What is it (2/2)?

Applications that deal with user input are particularly vulnerable to RCE. An example of such an application could be a spreadsheet application that takes user input as an Excel function, executes it on the server side and returns the result.

If the user input is not properly sanitised and sandboxed, an attacker will use it to execute malicious code and infect the server.

Remote Code Execution - Examples (1/3)

`eval()` is the most common way to execute dynamic code in JavaScript.



```
// Any kind of code can be placed inside eval function,
// including fetch API, DOM operations and more.

eval('alert(document.cookie)')
```

Remote Code Execution - Examples (2/3)

Dynamic code can also be executed by exploiting `Object()` and `Function()` constructors.

```
const object = {}
const a = 'constructor'
const b = 'alert(document.cookie)'

object[a][a](b)()
```

Remote Code Execution - Examples (3/3)

```
import { exec } from 'node:child_process'

app.get('/git-history', (req, res) => {
  const file = req.query.file
  const command = `git log --oneline ${file}`

  exec(command, (err, output) => {
    if (err) {
      res.status(500).send(err)
      return
    }
    res.send(output)
  })
})
```

❌ Never do this!

Remote Code Execution - How to prevent?

- Treat all user input as **untrusted** unless proven otherwise. Attackers will use any possible vector to inject custom code, it could be HTTP request headers, a random GET query parameter or form input.
- Anything you process should be sanitised first, and if you really need to execute it, it should be done in a sandboxed environment.

Open Redirects - What is it?

- Open redirect is when your web application redirects users to URLs taken from an untrusted, user-controllable, source.
- Open redirects can lead to phishing attacks, where a user sees what appears to be a legitimate URL, but is redirected to an attacker's website.
- Open redirects are often found in functions such as login, where your application redirects the user to a predefined URL after performing a login action. Such URLs typically look like `/login?next=/dashboard`

Open Redirects - Example

```
import * as express from 'express'
import * as fs from 'fs'
import * as path from 'path'

const app = express()

app.post('/login', (req, res) => {
    const {
      username, password, next,
    } = req.body

    // Simplified authentication check
    if (username === 'user' && password === 'pass') {
        res.redirect(next)
    } else {
        res.send('Invalid credentials')
    }
})
```

❌ Never do this!

This code checks the user's credentials and redirects them to the specified `next` URL.

It is vulnerable because the `next` parameter is user-controlled and can be any URL.

Open Redirects - Real life case

In 2018, a security researcher found an open redirect vulnerability on the Semrush website. A victim could be sent a URL such as https://www.semrush.com/redirect?url=http://bing.com, which would redirect them to any service.

This allowed the attacker to redirect victims to any website without them having to take any action (such as logging in).

Open Redirects - How to prevent (1/2)?

- Do not allow off-site redirects. Ensure that the URL provided by the user is relative (starting with `/`), not absolute.
- Another way to prevent open redirects is to check the `Referrer` HTTP header of requests. If the referrer is not your site, it means that the user came to your URL from another application (such as email) and usually indicates an exploit by an attacker.

Open Redirects - How to prevent (2/2)?

```
import * as express from 'express'

const app = express()

const isRelative = (url) => url && url.match(/^\/[^\/\\]/)

app.post('/login', (req, res) => {
    const {
      username, password, next,
    } = req.body

    // Simplified authentication check
    if (username === 'user' && password === 'pass') {
        if (isRelative(next)) res.redirect(next)
    } else {
        res.send('Invalid credentials')
    }
})
```

✅ Do this instead.

In your code, you can check if the URL provided is relative (points to a page on your site) and if so, redirect it.

Regex Denial of Service - What is it (1/2)?

Regular expressions are commonly used to validate texts, check their length, characters and so on.

```
const EMAIL_REGEX = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
const PASSWORD_REGEX = /^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d]{8,}$/;

app.post('/signup', (req, res) => {
    const { email, password } = req.body;

    if (!EMAIL_REGEX.test(email)) {
        return res.status(400).send('Invalid email format');
    }

    if (!PASSWORD_REGEX.test(password)) {
        return res.status(400).send('Password must be at least 8 characters long and
include both letters and numbers');
    }

    // ...
});
```

Regex Denial of Service - What is it (2/2)?

Regex is considered to be quite fast, even on long text or with complex regular expressions.
But a specially crafted regular expression can crash the server because it's exponentially computed.

`/(a|a)+$/` expression will cause a string like `aaaab` to be calculated exponentially. Each new a doubles the calculation time.

Regex Denial of Service - Real life case

Cloudflare used regex for their web application firewall (WAF) engine. The WAF allowed site administrators to detect and prevent various SQL injection, or XSS, attacks on their website.

However, in 2019, they released an update with a regex that contained a catastrophic backtracking problem that brought the entire company down for a while.

Regex Denial of Service - How to prevent?

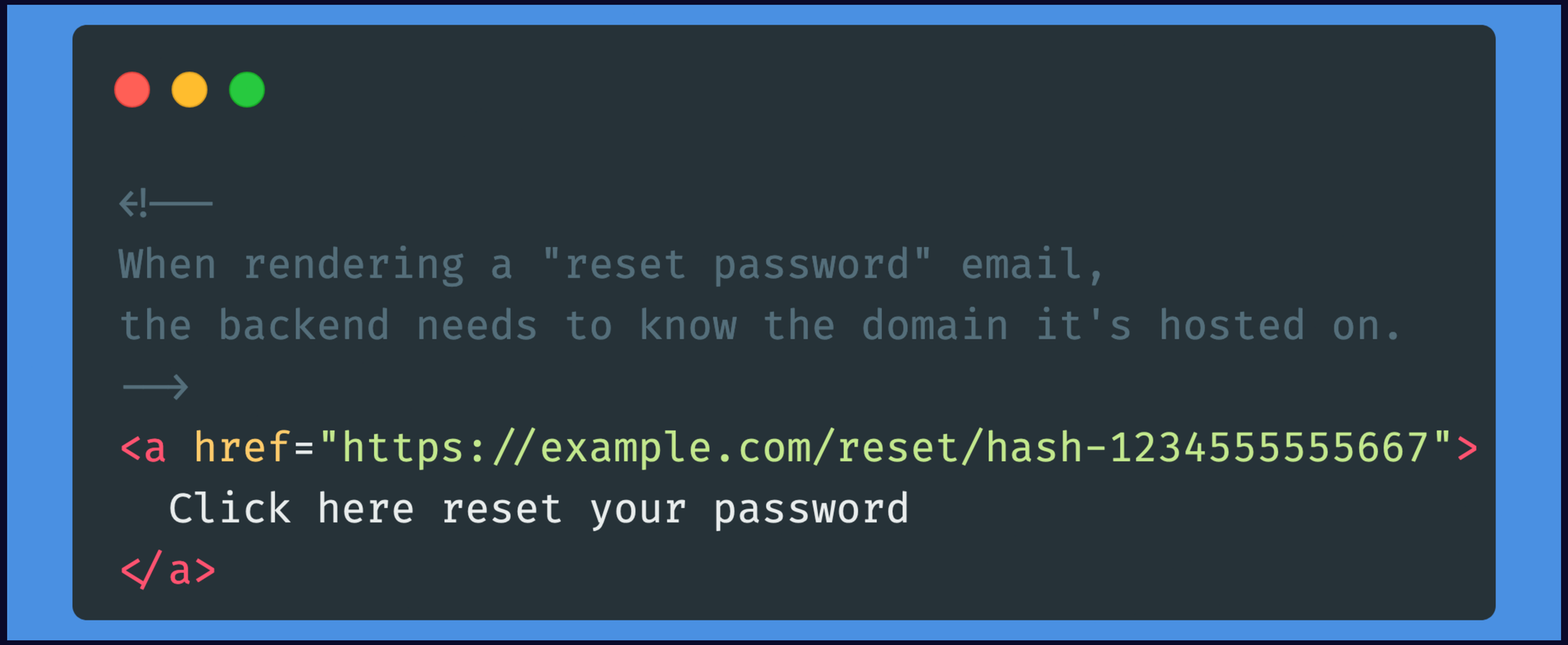
- Regular expressions should never be used from user input. They should be defined by the developers in the code base.
- If you absolutely must use user input as regex, consider using static analysis tools such as `safe-regex` to ensure that the input regular expression is safe from catastrophic backtracking.
- Consider switching to other regex engines, such as `RE2`, which guarantee linear regex execution.

Host Header Injections - What is it (1/3)?

- Web servers usually don't know which domain they are hosted on and are referred to by an IP address. DNS is responsible for finding a server IP address for a particular domain.
- This can be exploited by an attacker who can lie to the web server about what domain it's hosted on and gain some advantage.

Host Header Injections - What is it (2/3)?

Not knowing which domain server is hosted on isn't a problem in most cases, but sometimes it's necessary for things like password resets or web caching.



Host Header Injections - What is it (3/3)?

Browsers set `Host`, `Origin` and other headers automatically, but they are strictly informational as they can be easily overwritten by an attacker.

```
POST /request-password-reset HTTP/1.1
Host: attacker.com
[ ... ]

email=victim@example.org
```

```
<a href="https://attacker.com/reset/ ... ">
  Click here reset your password
</a>
```

Host Header Injections - Real life case

In 2023, a host header injection vulnerability was found on one of the subdomains of the US Department of Defense website.

This allowed a potential attacker to take over the subdomain and serve any content to users.

Host Header Injections - How to prevent?

- Use relative links instead of absolute ones whenever possible. It would prevent most cases of attacks like web cache poisoning.
- When crafting absolute URLs, do not rely on `Host` or `Origin` headers. Instead, use your earlier defined link from your configuration file.
- If you must use Host header, validate it. Create a list of permitted domains and check requests against it. Reject any requests for unknown hosts.

Vulnerable dependencies - What is it?

- Most developers don’t write software from scratch and use at least a few third-party dependencies to handle specific aspects of their applications, such as HTTP frameworks, parsing libraries, etc.
- It’s a good approach because it allows development teams to focus on their core product, but it also opens the door to vulnerabilities in these third-party packages, some of which may even be deliberately placed by their authors.

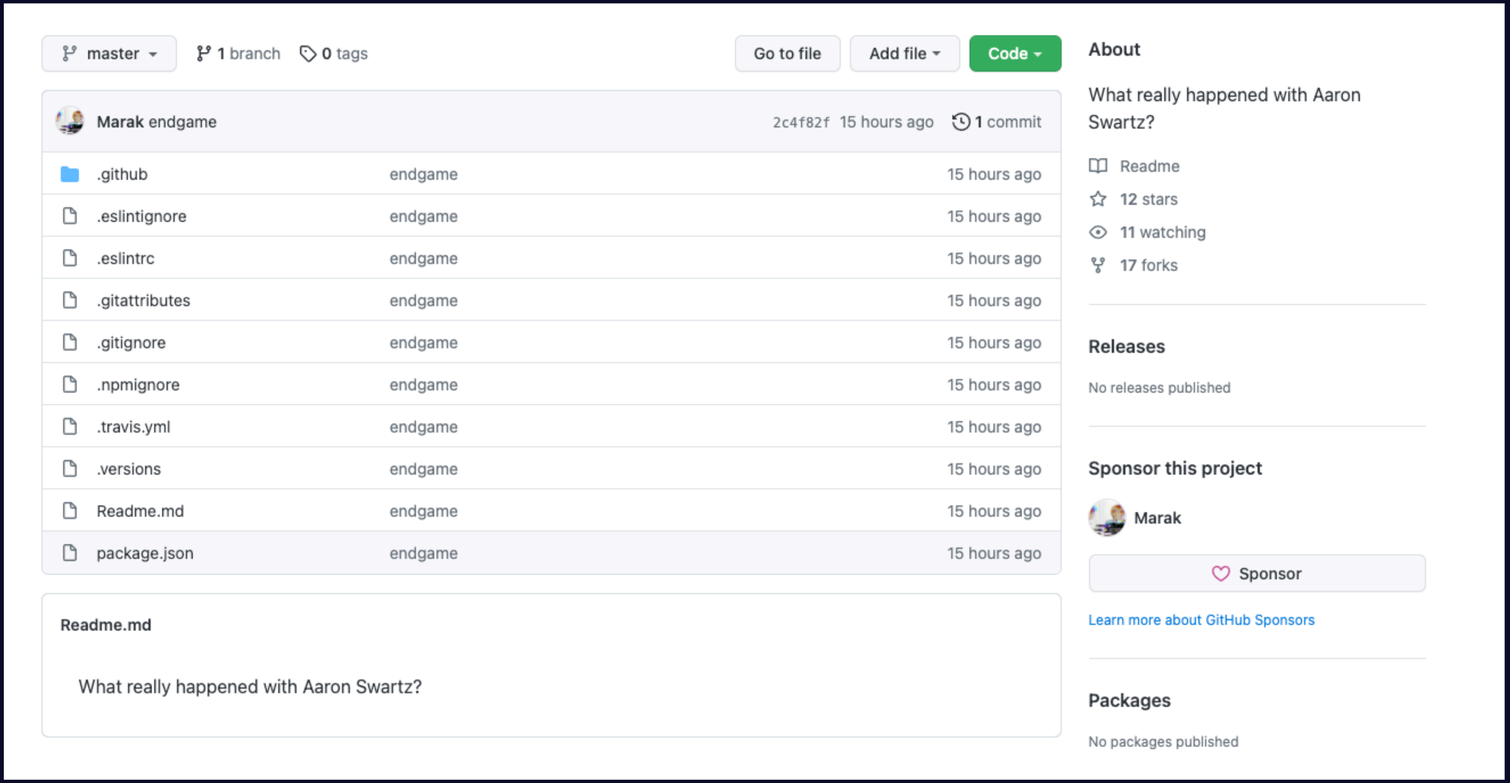
Vulnerable dependencies - Example (1/2)

Solarwinds attack

- In 2020, hackers inserted malicious code into updates of the SolarWinds Orion software. The compromised updates were signed with SolarWinds' legitimate digital certificate and distributed to18,000+ customers.
- Hundreds of organisations that relied on this software, such as the US Treasury, Intel or Microsoft, were compromised.

Vulnerable dependencies - Example (2/2)

faker.js



The author of the Faker.js library (4M+ downloads on NPM) deliberately released a broken version of his library.

It broke a lot of applications that depended on it, and a community maintained version had to be released.

Vulnerable dependencies - How to prevent?

- Use tools like Github Dependabot to periodically scan your dependency tree and alert you when something is wrong.
- Try to use popular software that has a large community behind it. This ensures that the project is analysed by thousands of people, and can be fixed quickly if something goes wrong.
- Use `package-lock.json` and pin your dependency versions to a known good version. Otherwise, the latest available version will be used, which could be malicious.

Thank you for your attention!