SERVER-SIDE CROSS-SITE SCRIPTING (SSXSS)

Balázs Bucsay

Founder & CEO of

Mantra Information Security

https://mantrainfosec.com

BIO / BALÁZS BUCSAY

- Over two decades of offensive security experience
- 15+ years of research and consultancy
- Software Reverse Engineer
- Started learning assembly at the age of 13
- Previously worked at NCC Group and Vodafone

- Certifications: OSCE, OSCP, OSWP; Prev: GIAC GPEN, CREST CCT Inf
- Frequent speaker on IT-Security conferences:
  - US - Washington DC, Atlanta, Honolulu
  - Europe - UK, Belgium, Norway, Austria, Hungary...
  - APAC - Australia, Singapore, Philippines

BIO / BALÁZS BUCSAY

- Happy to chat! Find me after the talk
- Hobbies:
  - Travelling (been to 75+ countries)
  - Hiking, kayaking, cycling
  - IT Security
- Passionate about learning from others
- Kayaked the length of the Thames (300km)

- Twitter: @xoreipeip
- Linkedin: https://www.linkedin.com/in/bucsayb/
- Mantra on Twitter: @mantrainfosec
- Mantra: https://mantrainfosec.com

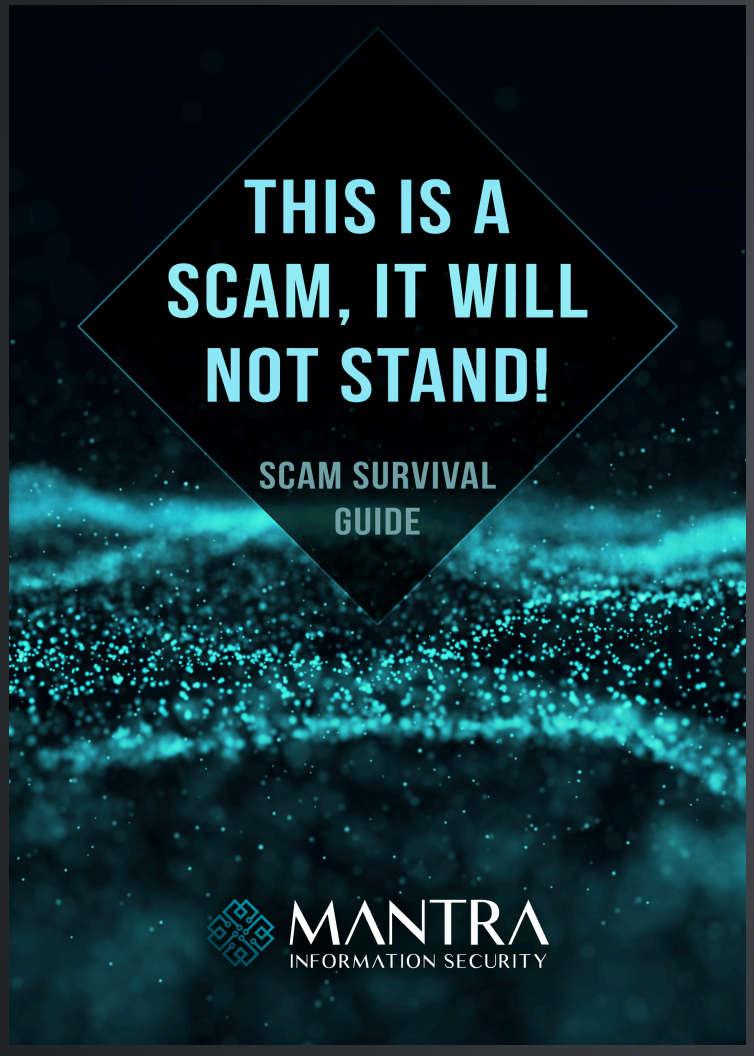
BIO / BALÁZS BUCSAY

MANTRA INFORMATION SECURITY

- London based boutique consultancy
- Decades of experience and excellence
  - Specialised training delivery (Software Reverse Engineering...)
  - Cloud, CI/CD, Kubernetes reviews
  - Red Teaming, EASM, Infrastructure testing
  - Web application and API assessments
  - Reverse-engineering, embedded devices and exploit development
  - ...
- Full stack consultancy: from identifying vulnerabilities to implementing fixes

https://MantraInfoSec.com

BOOK: THIS IS A SCAM! IT WILL NOT STAND!



- Empower others to protect themselves
- 11 gripping and educational real-world stories
- Download and share it freely
- Absolutely free of charge
- Perfect for a non-technical audience

https://mantrainfosec.com/scambook

SERVER-SIDE CROSS-SITE SCRIPTING

- That is a mouthful
- Let's unfold it a bit

CROSS-SITE SCRIPTING

Most well-known and misundertood payload:

alert(1)

CROSS-SITE SCRIPTING

- Type of an injection attack
- Injection of malicious HTML or JavaScript into a web page
- Different types including:
  - Stored (Persistent)
  - Reflected
  - DOM-Based
- Always executed in the user's browser
- Always?

ROOT CAUSES OF CROSS-SITE SCRIPTING

- Improper input validation
- Improper input sanitization
- Improper output sanitization

- Choose your poison

IMPACT OF CROSS-SITE SCRIPTING

- Impact:
  - User impersonation
  - Account hijacking
  - Website defacement
  - Phishing attacks
  - ...
- It's often underrated:
  - Considered "trivial" to find
  - Easy to misunderstand or underestimate

- One thing is sure, it only affects the client (browser)
- Or does it?

JAVASCRIPT

- Purposefully chose the name Server-Side XSS instead of Server-Side JS Injection
- JavaScript is everywhere now:
  - Web Applications
  - Web Servers - Think NodeJS
  - Desktop Applications - Think Electron
  - ... and beyond

JAVASCRIPT INJECTION

- JavaScript injection doesn’t just affect browsers anymore
- Improperly handled user-input can lead to:
  - XSS in Web Applications
  - Remote Command Execution in Web Servers (NodeJS)
  - Remote Command Execution in Desktop Applications (Electron)
- We won't cover these scenarios today

SERVER-SIDE CROSS-SITE SCRIPTING

If XSS runs in the client's browser how can it be Server-Side?

SERVER-SIDE CROSS-SITE SCRIPTING

If XSS runs in the client's browser how can it be Server-Side?

Because the browser runs on the server!

DIFFERENT DELIVERY APPROACHES

- Move fast and break things + Diverse technology stacks
  = sloppy implementations an overlooked security risks
- Think of PDF generation on server side
  - In the past: a SW library that created PDF from text
  - Now: Docker + Chrome + HTML = PDF
- Which approach is better? Not sure
- Which approach is more secure?
  - The faster pace often leads to less testing and a lack of security awareness

DEMO ENVIRONMENT REQUIREMENTS

- Let’s walk through a scenario:
  - A web application is running on AWS
  - Users require a feature to export reports in PDF format
  - Docker containers (via ECR/ECS/Fargate), are used to handle the PDF rendering
  - The rest of the environment and infrastructure is hidden from the user

HTML 2 PDF

- Straightforward solution is to use a browser to convert HTML to PDF
  - Let's get a Docker container that does just this:
    - One of many: export-html
  - Provides an API over HTTP
  - HTML is POST'd and PDF is returned
  - What could go wrong with it?

THE ENVIRONMENT

- A test environment was ~~replicated~~ built
- Simple steps to achieve it:
  - IAM roles and policies configured
  - VPC, Subnet, Routing table, and an Internet Gateway added
  - Two docker containers deployed using ECS with Fargate
  - Security groups configured to control network access

TWO CONTAINERS

- To make things more interesting, two docker containers were added:
  - export-html: renders PDF from HTML
  - custom built container: intentionally vulnerable to LFR + SSRF
- Local File Read (LFR): Can read arbitrary files from the container's filesystem
- Server-Side Request Forgery (SSRF): Can interact with internal services on the network

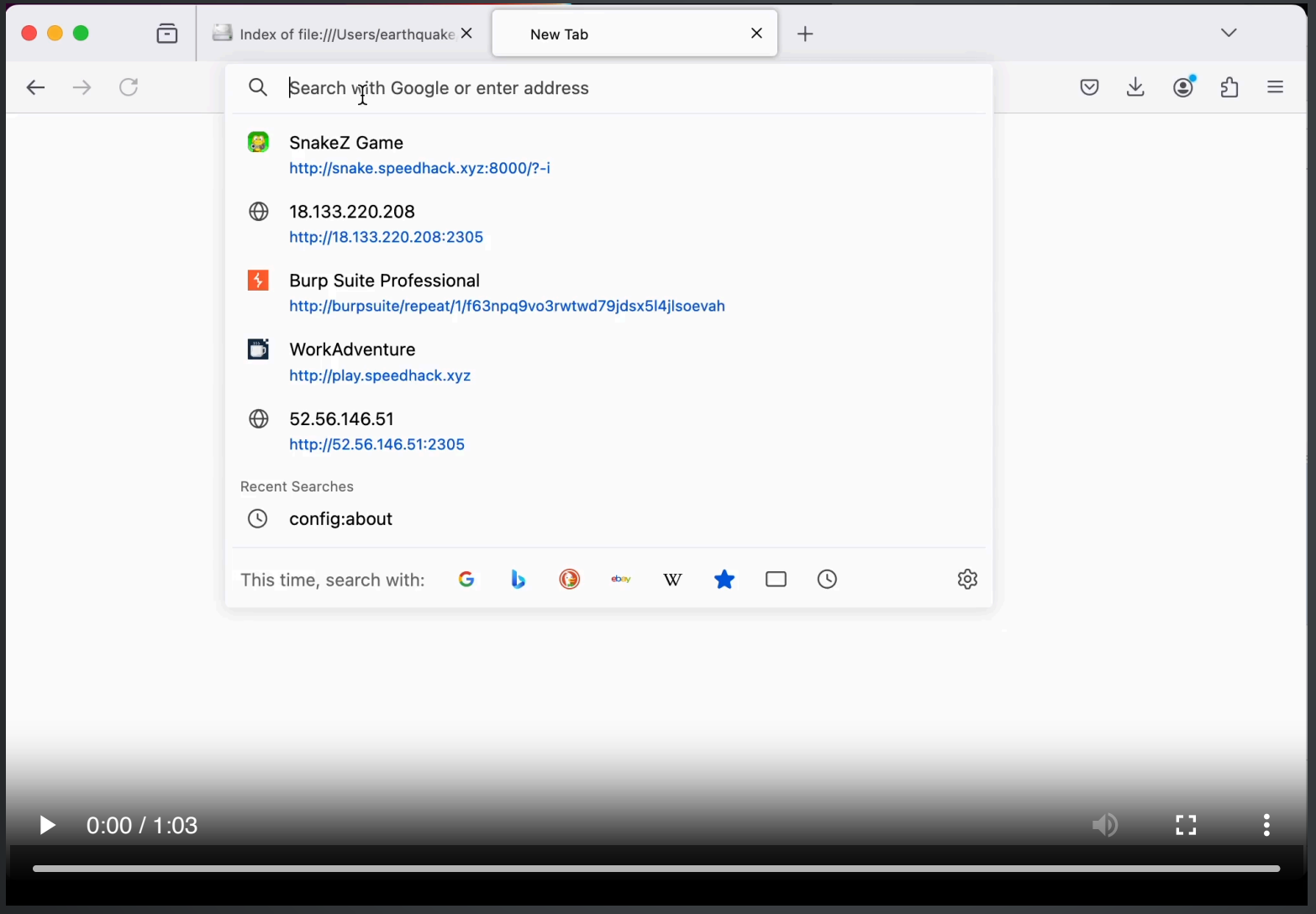
RENDERING DEMO

HOW DOES THIS WORK?

- HTML content is sent in JSON in a POST request to the API
- The server uses Headless Chrome to render the HTML
- The rendered output is returned to the user in PDF format

- The server runs a real browser to render the page
- If the site is vulnerable to XSS => Server-Side XSS

HOW DOES THIS WORK?



EXTERNAL CONNECTION

XSS OR NOT?

- Cross-Site Scripting only works if JavaScript is enabled
- Let's check for JS is available in the browser

```
{"html": "<script>document.write('<h1>'+
(33074-1737).toString()+'</h1>');</script>"}
```

XSS OR NOT?

XSS

- Now that we know XSS is possible
- What can we do with JavaScript or HTML?
  - Interact with other services over the network:
    - Use SMB to relay or steal hashes
    - Leverage SSRF to attack internal services or retrieve internal details
    - Port scan the network to identify open ports
    - Access cloud-related metadata or credentials
  - Read local files (dependent on Same-Origin Policy)
  - Launch Denial of Service (DoS) attacks

SMB INTERACTION

- Only available on Windows boxes
- Open SMB share from HTML/JS: \\external.IP\test\test
- Use Responder on your side (external IP needed)
- Domain credentials might appear, depending on the setup

- We are using Linux container, so this doesn't apply

LOCAL FILE READ

- Subject to Same Origin Policy (SOP)*
- Works only when the payload is written to a file and opened for rendering
- Check: window.location for the URL
- about:blank does not count as a file

- No Local File Read for us! At least, not from the browser

* https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

DENIAL OF SERVICE

- Overloading the service:
  - Huge requests to overwhelm the server
  - Recursive content (iframe within an iframe)
  - Infinite or recursive loop
- This could render the service unavailable or result in extra charges for the owner

- We do not plan to cause harm

CLOUD METADATA

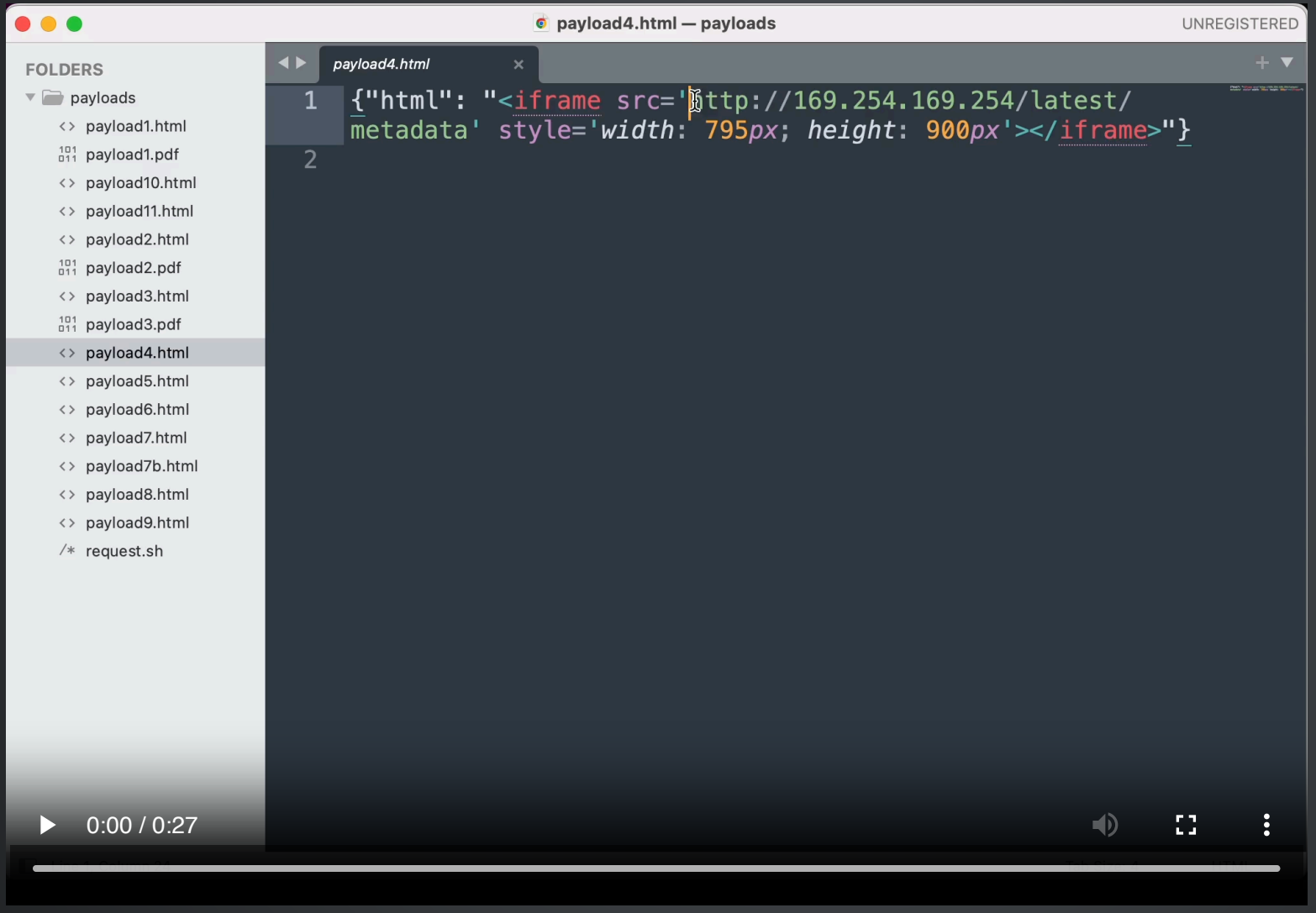
- Metadata host: http://169.254.169.254/
- This seems universal across cloud providers
- Metadata contains instance-related data
- It might include credentials (e.g. access tokens)
- Can be access over HTTP with or without an extra header

AWS METADATA

- IMDSv1 - a simple request/response method (old) - can be rendered from browser
- IMDSv2 - a session-oriented method requiring extra headers:
  - X-aws-ec2-metadata-token-ttl-seconds (PUT)
    - Can't be sent from HTML
    - Can be sent from JS, but no CORS headers - so it won’t render
  - X-aws-ec2-metadata-token (GET)
    - Requires the token in header

- This would be relevant for us if the container was running on EC2

AWS METADATA



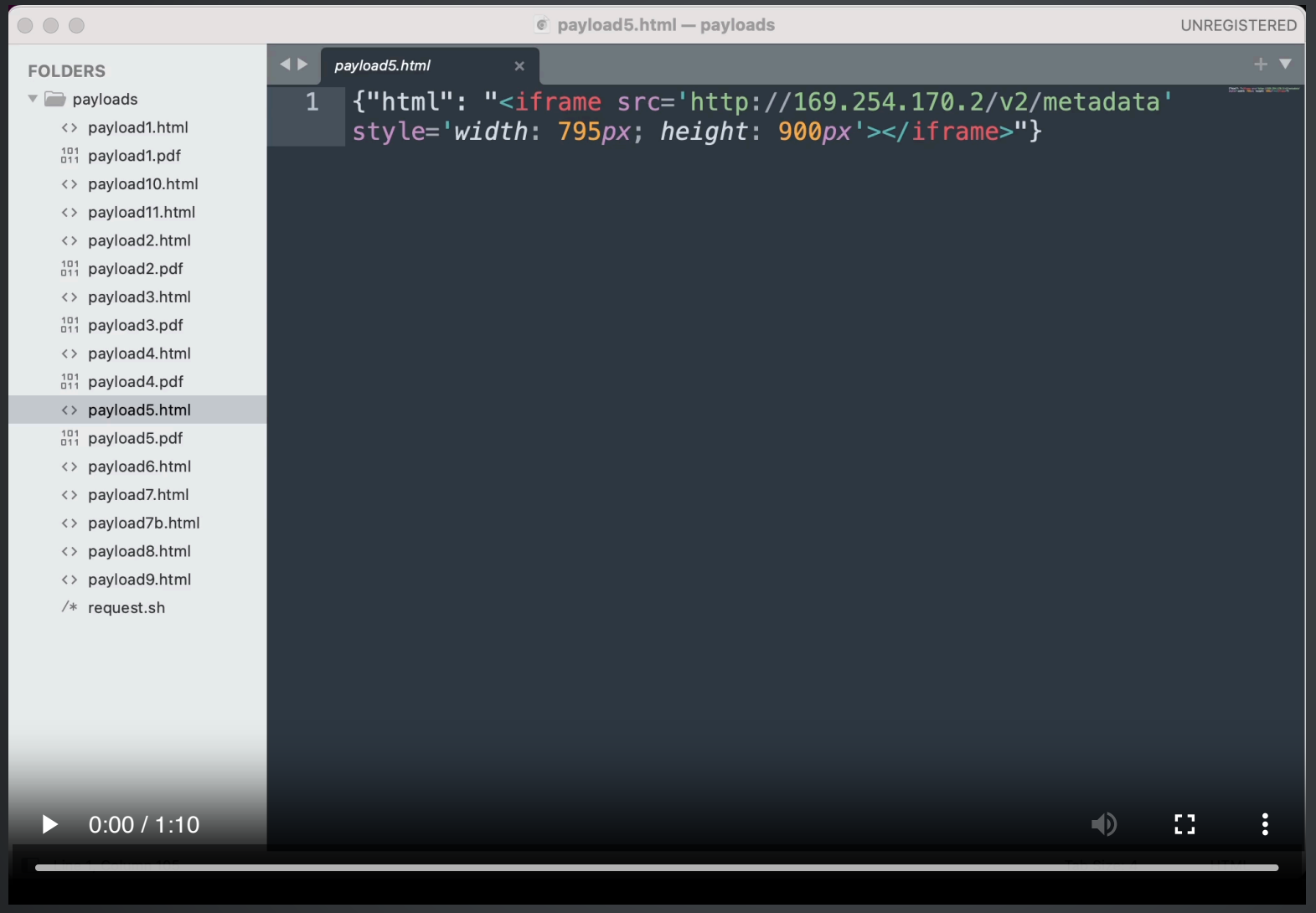
GCP/AZURE METADATA

- Metadata host: http://169.254.169.254/ (metadata.google.internal)
- GCP extra header: Metadata-Flavour: Google
- Azure header: Metadata: true
  - No way to add header from HTML
  - No CORS headers in response

AWS METADATA ON ECS

- Metadata host: http://169.254.170.2/
- To access the juicy data, we would need the container's GUID
  - http://169.254.170.2/v2/credentials/<GUID>
- GUID can be found under: /proc/self/environ
- Local File Read is required to access the file content

AWS METADATA ON ECS

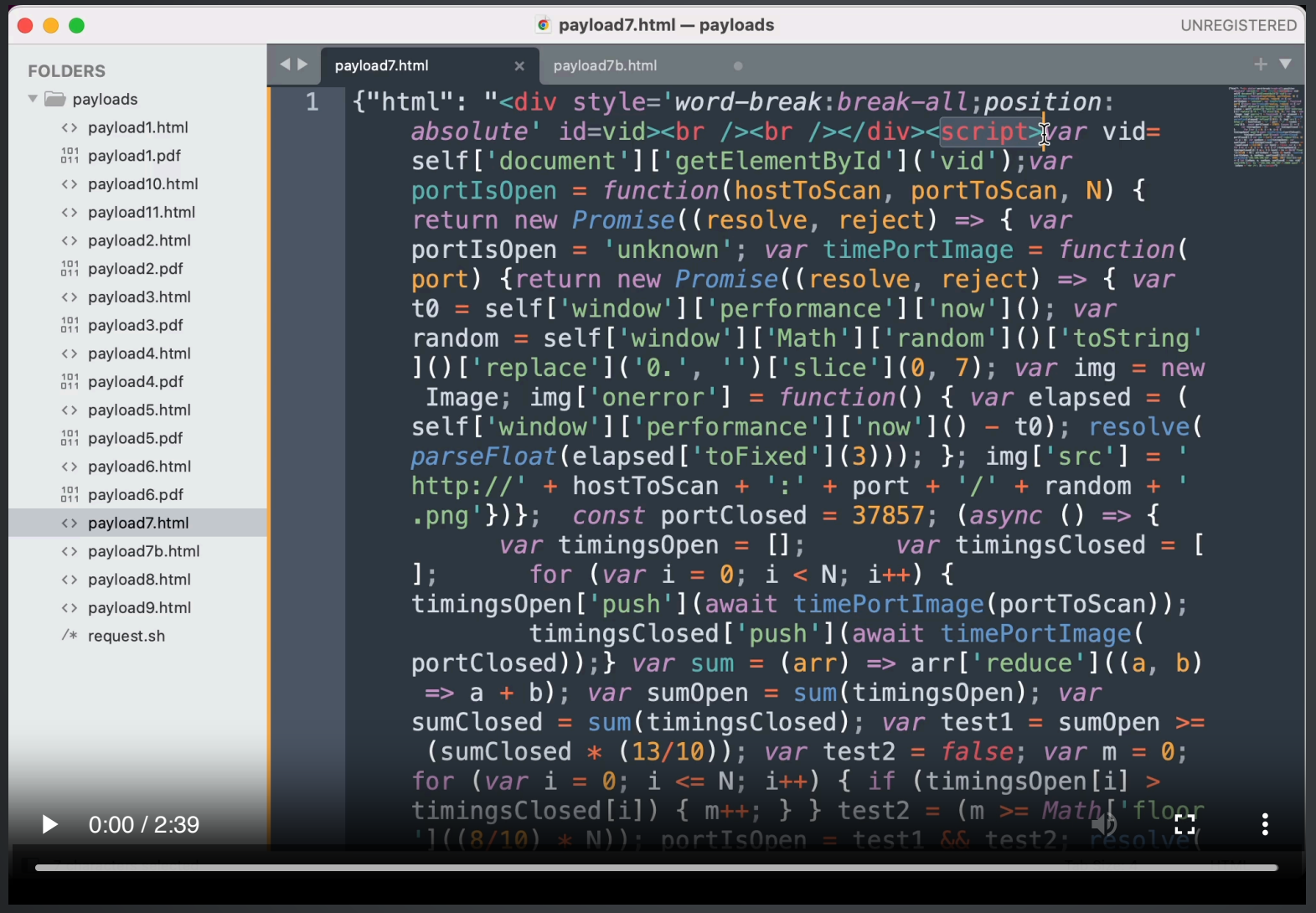


PORT SCAN

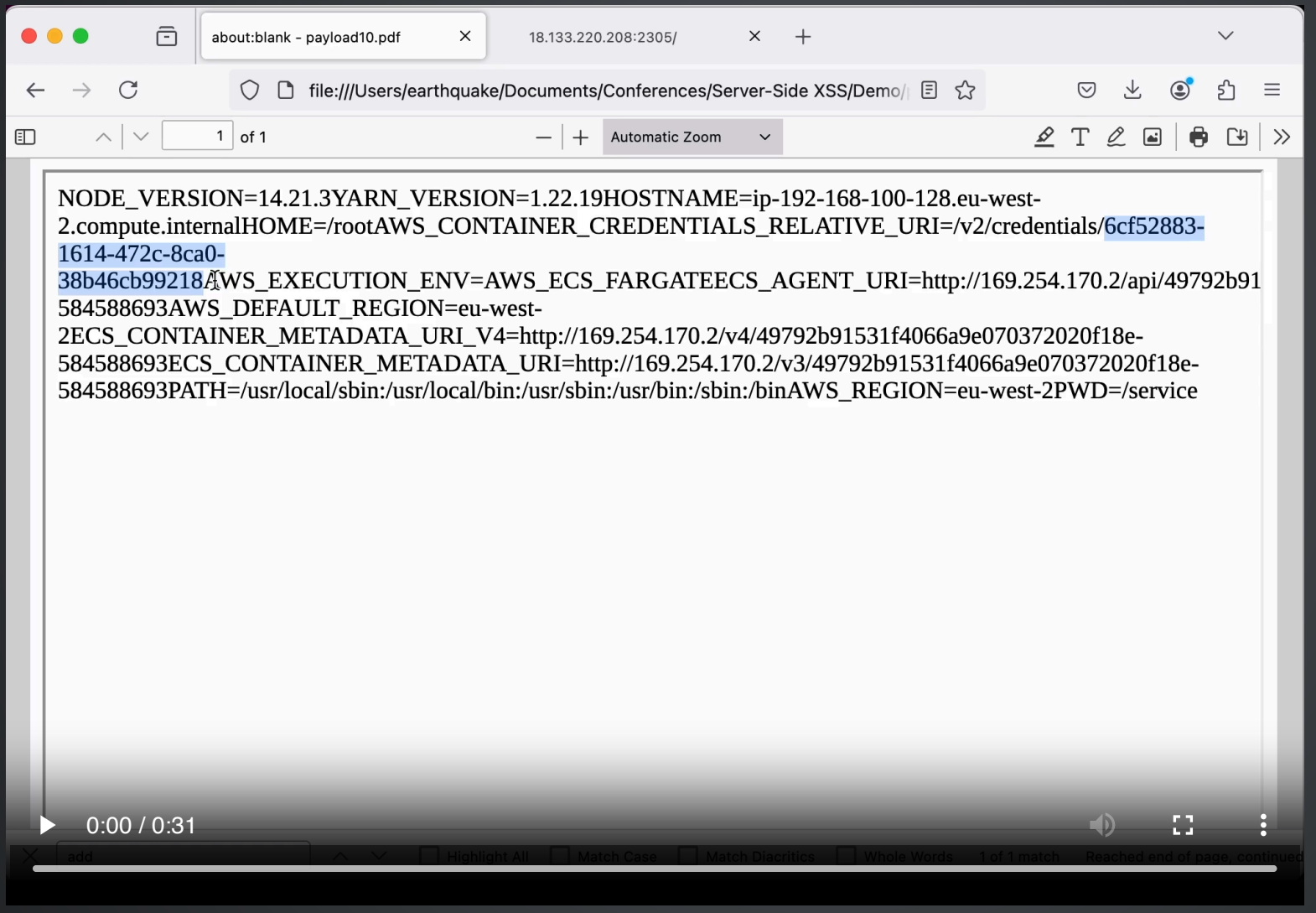
- Somewhat possible from JavaScript
- Cool research on the topic (Nikolai Tschacher):
  - https://incolumitas.com/2021/01/10/browser-based-port-scanning/
- WebSocket/IMG methods are detailed in the article above
- These methods rely on timing, which makes it somewhat unreliable
- Hosts that are non-existent or firewalled do not send RST packets

- Oldschool ways:
  - Check for timeout
  - Open in an iframe if the port is HTTP(S)

AWS METADATA ON ECS & PORT SCAN



CREDENTIALS FROM AWS METADATA



HTML 2 PDF

Cute security note

Export HTML to PDF Service

This is a simple Docker container that runs a JSON API service that allows HTML to be converted to PDF or PNG/JPG images. This service accomplishes this by using a Chrome headless browser to ensure full rendering capabilities on par with Google Chrome.

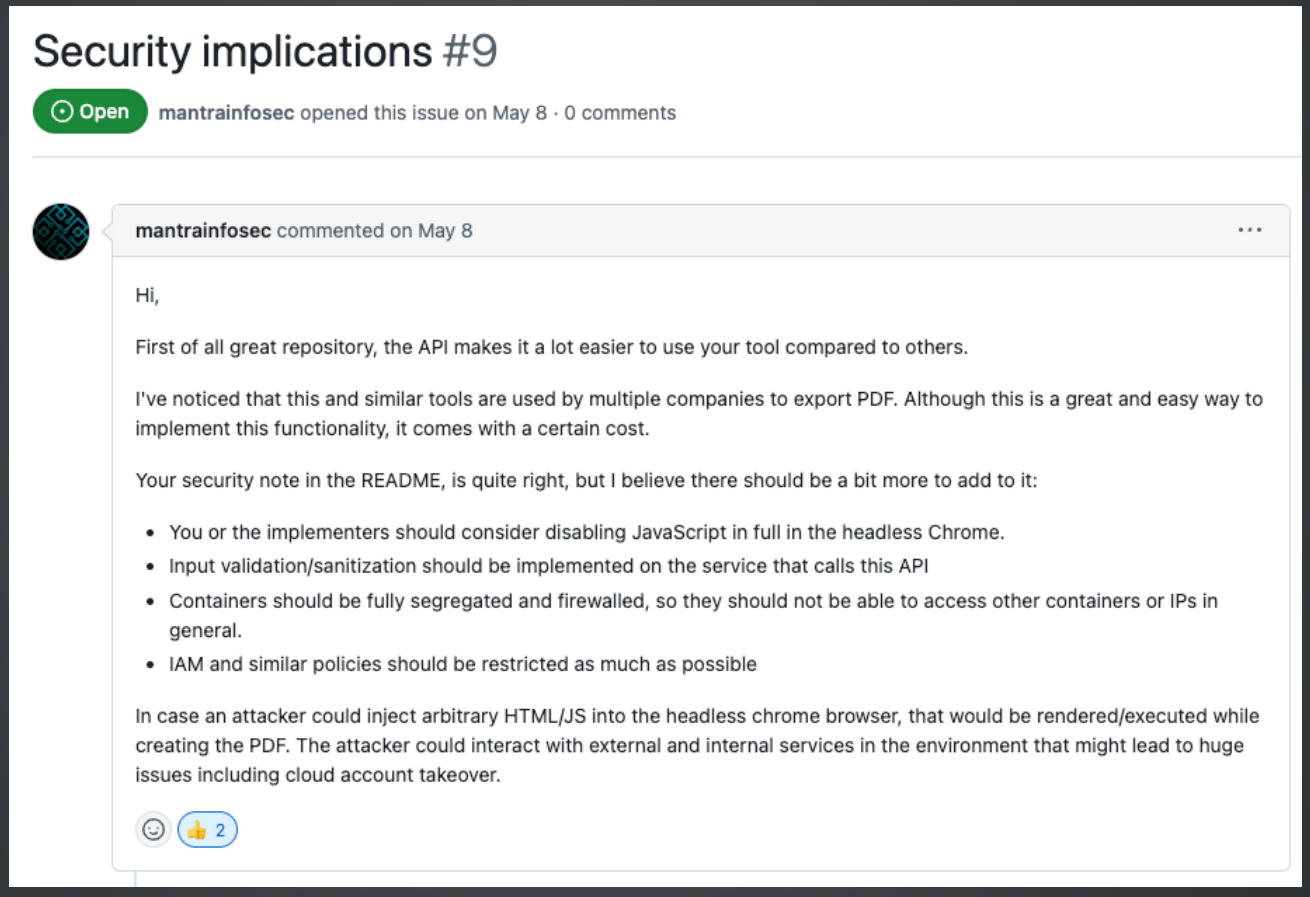
Security Note: This is intended to run as a micro service - do not directly expose to the public internet

HTML 2 PDF

- It's worth noting — that warning is absolutely valid!
- Also a few additional things to consider:
  - Disable JavaScript
  - Run it in a fully isolated/firewalled environment
  - Validate and sanitze all input before processing it

- Otherwise... what could go wrong?

HTML 2 PDF - ISSUE OPENED



Security implications #9

Open **mantrainfosec** opened this issue on May 8 · 0 comments

mantrainfosec commented on May 8

Hi,

First of all great repository, the API makes it a lot easier to use your tool compared to others.

I've noticed that this and similar tools are used by multiple companies to export PDF. Although this is a great and easy way to implement this functionality, it comes with a certain cost.

Your security note in the README, is quite right, but I believe there should be a bit more to add to it:

- You or the implementers should consider disabling JavaScript in full in the headless Chrome.
- Input validation/sanitization should be implemented on the service that calls this API
- Containers should be fully segregated and firewalled, so they should not be able to access other containers or IPs in general.
- IAM and similar policies should be restricted as much as possible

In case an attacker could inject arbitrary HTML/JS into the headless chrome browser, that would be rendered/executed while creating the PDF. The attacker could interact with external and internal services in the environment that might lead to huge issues including cloud account takeover.

2

FIX / MITIGATION: CODING

- Let's start from the beginning
- Always implement at least one of the following:
  - Input validation
  - Input sanitization
  - Output sanitization
- Not just for XSS, any user input should be treated this way

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

FIX / MITIGATION: HIGH LEVEL

- Thread model: Plan before implementing anything new
  - Think through what you are going to build
  - Consider different angles and potential attack vectors
  - Ask yourself: How would YOU abuse it?
- Least Privilege Principle & Separation of Duties
  - Grant only the necessary privileges—the fewer, the better
  - Avoid sharing users between separate functions
- Defense in Depth
  - Segregated network for better security
  - Strict firewall rules to control access

FIX / MITIGATION: HTML 2 PDF

- Reconsider using this technology
- If it's the only way forward, take these precautions:
  - Disable JavaScript in the browser
  - Configure the browser to be as secure as possible
  - Run it in a fully isolated/firewalled environment
    - No network or Internet access
    - Render pure HTML with embedded pictures. Nothing else
  - Ensure all input is validated and sanitized (sounds familiar?)

MANTRA

INFORMATION SECURITY

Q&A

https://mantrainfosec.com


MANTRA
INFORMATION SECURITY