SERVER-SIDE CROSS-SITE SCRIPTING (XSS)

Balázs Bucsay

Founder & CEO of

Mantra Information Security

https://mantrainfosec.com

BIO / BALÁZS BUCSAY

- Over two decades of offensive security experience
- Started learning assembly when 13 years old
- Reverse engineering software and modifying them
- 15 years of research and consultancy
- 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
- Love to learn from others

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

MANTRA INFORMATION SECURITY

- Boutique consultancy approach
- Decades of experience and excellence
  - Training delivery (Software Reverse Engineering Training)
  - 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 finding a bug until it gets fixed

https://mantrainfosec.com

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
- Malicious HTML/JavaScript content injected into a page
- Different types including:
  - Stored (Persistent)
  - Reflected
  - DOM Based
- Always gets rendered/executed in the client's browser
- Always?

ROOT CAUSE 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 can be underrated because:
  - "Trivial" to find
  - Easy to misunderstand

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

JAVASCRIPT

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

JAVASCRIPT INJECTION

- JS injection does not only affect the browser 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 are not covering these today

SERVER-SIDE CROSS-SITE SCRIPTING

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

SERVER-SIDE CROSS-SITE SCRIPTING

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

Because the browser runs on the server!

DIFFERENT DELIVERY APPROACHES

- The need to deliver fast, demands new solutions
- These combined with the diverse technology create sloppy implementations
- Think of PDF generation on server side
  - In the past: coded a SW library that creates PDF from text
  - Now: Docker + Chrome + HTML = PDF
- Which approach is better? Not sure
- Which approach is more secure?
  - Faster pace comes with less testing and lack of security awareness

DEMO ENVIRONMENT REQUIREMENTS

- Let's see a scenario where:
  - Web application running on AWS
  - Users need a functionality to export reports in PDF format
  - Docker with ECR/ECS/Fargate is used to render the reports
  - Other parts of the environment are 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 can be POST'd and PDF is returned
  - What could go wrong with it?

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.

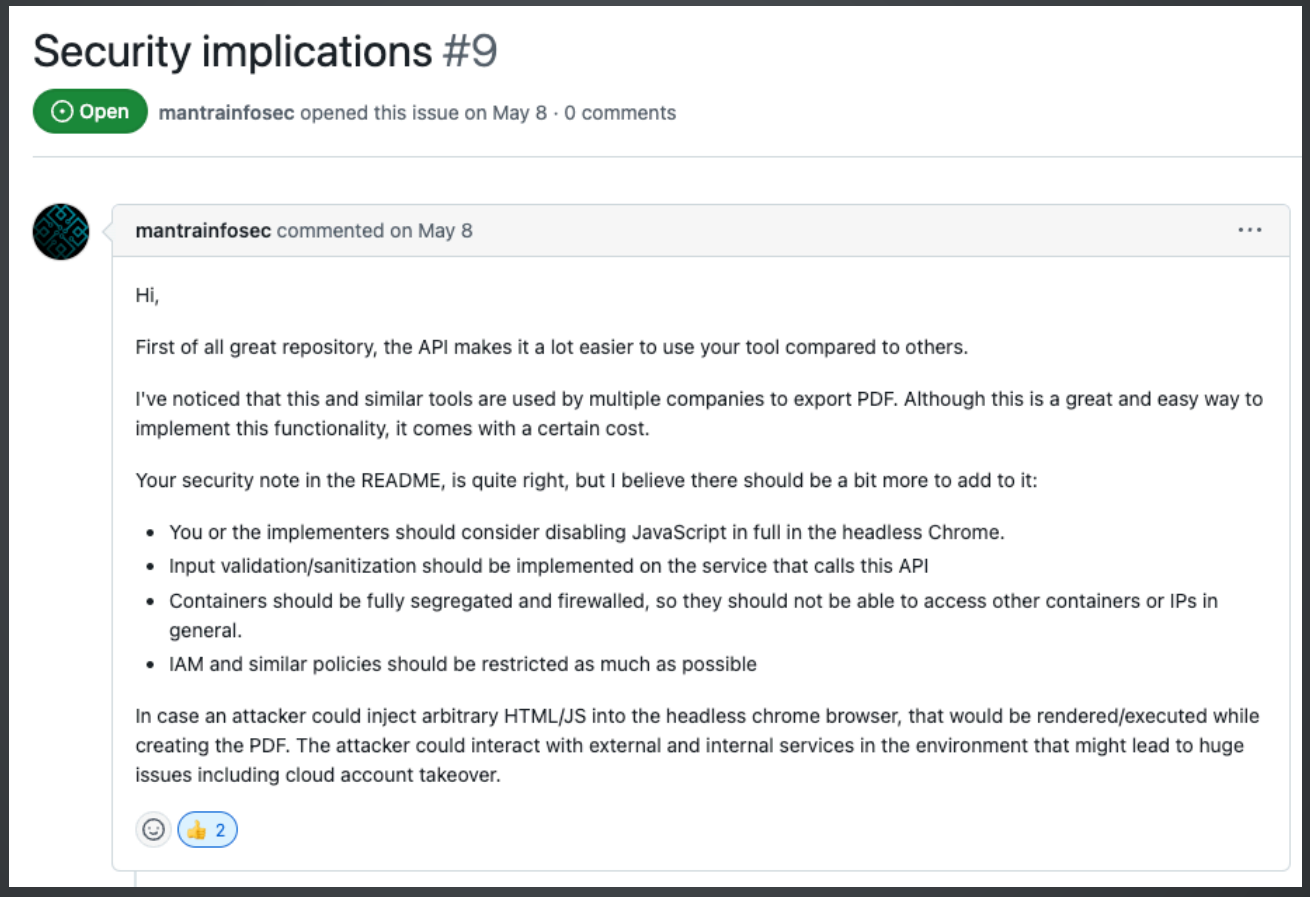
Security Note: This is intended to run as a micro service - do not directly expose to the public internet

HTML 2 PDF

- Must be mentioned, that the note is right!
- Also a few other things to add:
  - Consider disabling JavaScript
  - Consider running it in a fully isolated/firewalled environment
  - Make sure that all input is validated and sanitized

- 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

THE ENVIRONMENT

- A test environment was ~~replicated~~ built
- Simple steps to achieve it:
  - IAM policies and roles added
  - VPC, Subnet, Routing table, Internet Gateway added
  - Two docker containers created in ECS/Fargate
  - Security groups created

TWO CONTAINERS

- To make this more interesting, two docker containers were added
  - export-html: to render PDF from HTML
  - custom built: vulnerable for LFR + SSRF
- Local File Read (LFR): Reads any file from the container's FS
- Server-Side Request Forgery (SSRF): Interacts with other services on the network

RENDERING DEMO

HOW DOES THIS WORK?

- HTML content in JSON posted to the API
- The HTML content is rendered in (Headless) Chrome
- Output/rendered HTML is sent back to the user in PDF format

- The server side has an actual browser and opens webpages
- XSS in that webpage => Server-Side XSS

XSS OR NOT?

- Cross-Site Scripting only works if JavaScript can be used
- Let's check for JS in the browser

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

XSS

- Now we know that XSS is possible
- What can we do from JavaScript or HTML?
  - Interact with other services over the network
    - SMB interaction to relay/steal hashes
    - SSRF to attack other services/get internal details
    - Port scan the network
    - Get cloud related metadata/credentials
  - Read local files (SOP dependent)
  - Denial of Service (DoS)

SMB INTERACTION

- Only on Windows boxes
- Open SMB share from HTML/JS: \\external.IP\test\test
- Use Responder on your side (external.IP)
- Domain creds might be popping up - depending on setup

- Not in our case, we have Linux containers

LOCAL FILE READ

- Subject to Same Origin Policy (SOP)*
- Works only when payload is written into a file and opened for render
- Check: window.location for 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
  - Recursive content (iframe in an iframe)
  - Infinite loop/Recursive loop
- It might render it unavailable or result in extra charge for the owner

- We do not plan to cause harm

CLOUD METADATA

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

AWS METADATA

- IMDSv1 - a request/response method (old) - can be rendered from browser
- IMDSv2 - a session-oriented method with 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 - no render
  - X-aws-ec2-metadata-token (GET)
    - Requires the token in header

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

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 get 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 needed to access the file content

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 detailed in the article above
- Based on timing, which makes it somewhat unreliable
- Non-existent/firewalled hosts do not send RST

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

FIX / MITIGATION: CODING

- Lets start from the beggining
- Always do at least of the three:
  - Input validation
  - Input sanitization
  - Output sanitization
- Not just for XSS, any user input qualifies

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

FIX / MITIGATION: HIGH LEVEL

- Thread model: Do it before implementing something new
  - Think through what you are going to implement
  - Think of different angles and attack vectors
  - How would YOU abuse it?
- Least Privilege Principle & Separation of Duties
  - Less privileges the better
  - Do not share users between separate functions
- Defense in Depth
  - Segregated network
  - Strict firewall rules

FIX / MITIGATION: HTML 2 PDF

- Reconsider using this technology
- If the only way forward:
  - Consider disabling JavaScript in browser
  - Configure the browser as secure as possible
  - Consider running it in a fully isolated/firewalled environment
    - No network or Internet access
  - Render pure HTML with embedded pictures and that is it
- Make sure that all input is validated and sanitized (remember?)

FIX / MITIGATION: LOCAL FILE READ

- Make sure other services do not present risk
- Local File Read vulnerabilities for example
- List goes on...

Q&A

https://mantrainfosec.com