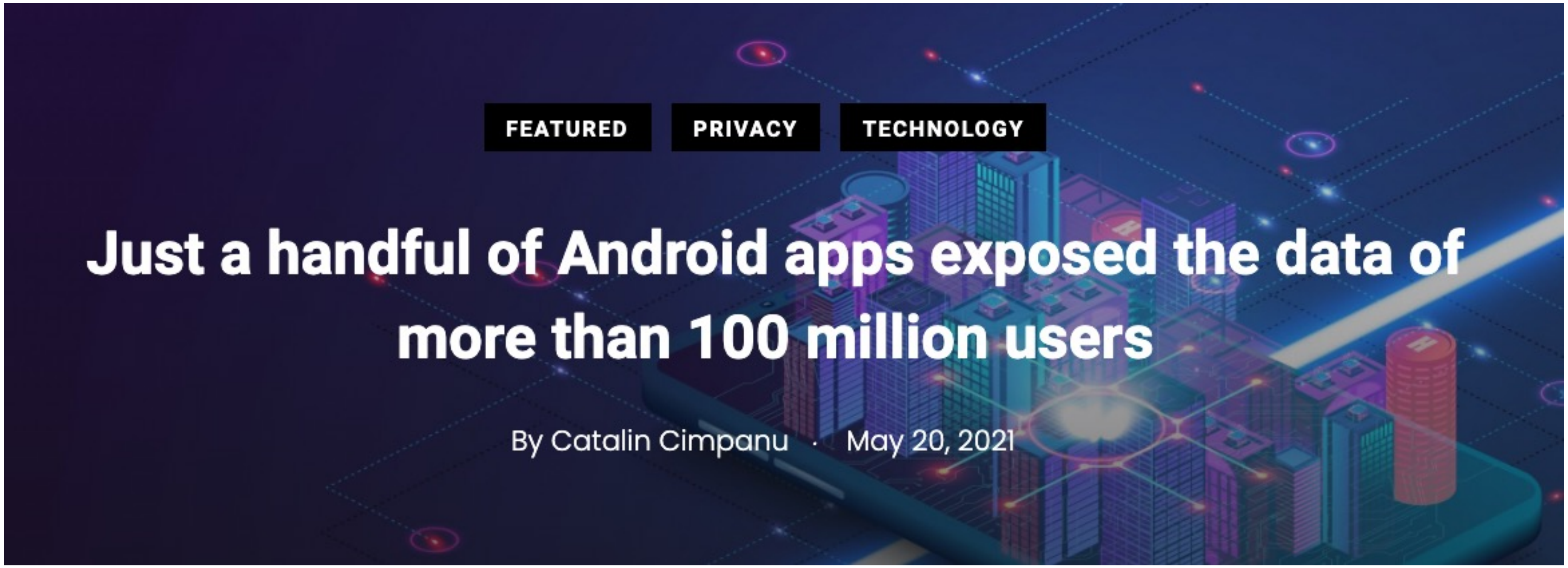
API security is too hard!?

Dr. Philippe De Ryck

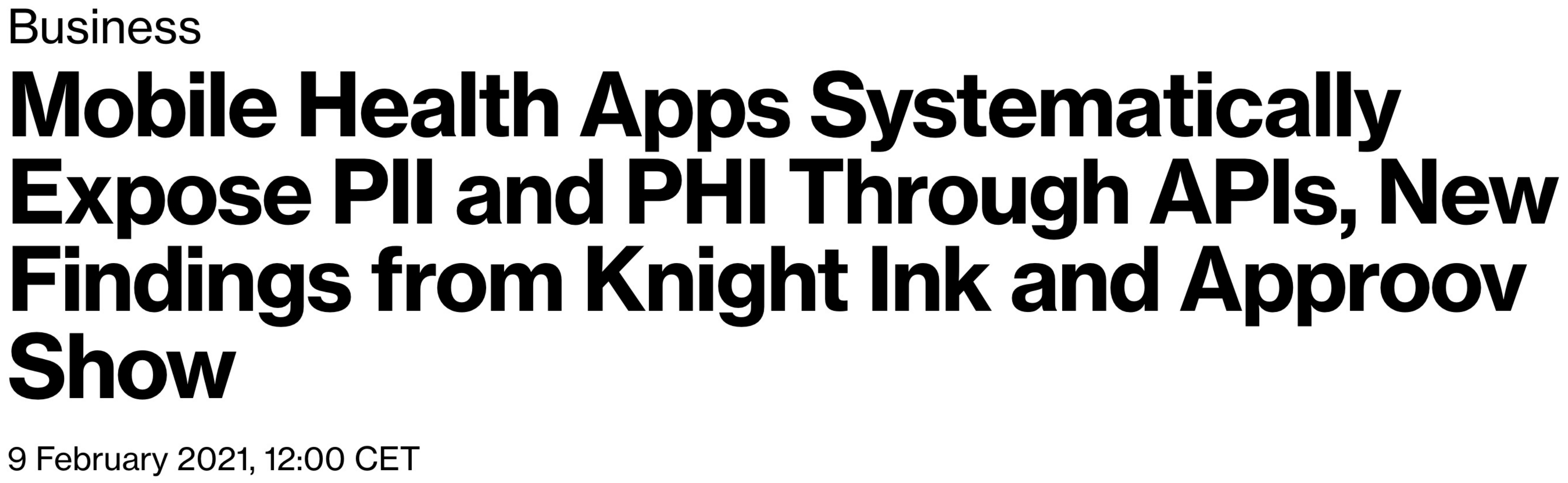

FEATURED PRIVACY TECHNOLOGY

Just a handful of Android apps exposed the data of more than 100 million users

By Catalin Cimpanu · May 20, 2021

https://therecord.media/just-a-handful-of-android-apps-exposed-the-data-of-more-than-100-million-users/

“The Check Point team said it was able to use the information they found through a routine examination of 23 random applications and access the backend databases of 13 apps.”



Business

Mobile Health Apps Systematically Expose PII and PHI Through APIs, New Findings from Knight Ink and Approov Show

9 February 2021, 12:00 CET

https://www.bloomberg.com/press-releases/2021-02-09/mobile-health-apps-systematically-expose-pii-and-phi-through-apis-new-findings-from-knight-ink-and-approov-show

“ **Of the 30 popular apps Knight Ink tested, 77 percent contained hardcoded API keys, some which don’t expire, and seven percent contained hardcoded usernames and passwords.** ”

Start taking security seriously

The cowboy years are over. Security is a crucial requirement for every application from day 1, and not an afterthought for a quiet period.

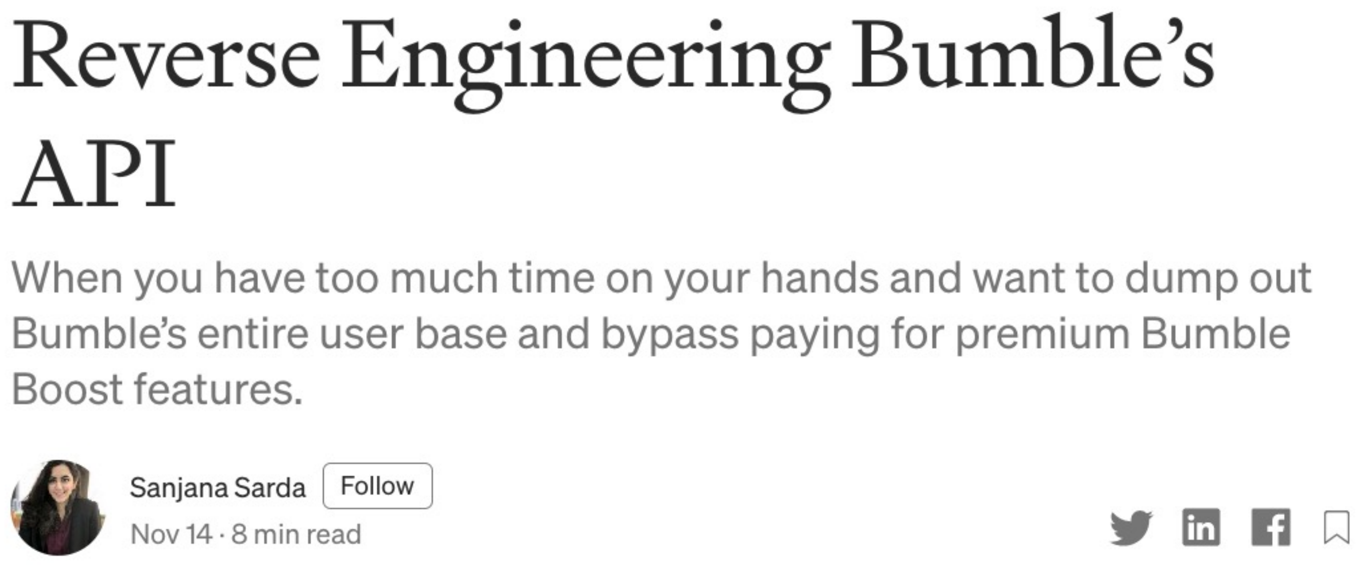
I am *Dr. Philippe De Ryck*

- Founder of Pragmatic Web Security
- Google Developer Expert
- Auth0 Ambassador
- SecAppDev organizer

I help developers with security

- ✓ Hands-on in-depth security training
- ✓ Advanced online security courses
- ✓ Security advisory services

https://pragmaticwebsecurity.com



Reverse Engineering Bumble's API

When you have too much time on your hands and want to dump out Bumble's entire user base and bypass paying for premium Bumble Boost features.

Sanjana Sarda Follow

Nov 14 · 8 min read

"Our accounts eventually got locked and hidden for more verification requirements. We tested retrieving user data while our account was locked, and it still worked."

https://blog.securityevaluators.com/reverse-engineering-bumbles-api-a2a0d39b3a87

Do not rely on client-side authorization

- Client-side authorization is common in API-based applications
    - Frontend applications typically need authorization information to configure the UI
    - Frontends deny access to features, hiding the ability to access prohibited API endpoints
    - As a result, many APIs consider these endpoints unreachable and fail to protect them
- The attack surface of an API is the set of accessible endpoints
    - The frontend ***does not play any role*** in the authorization process
    - Sequential steps / transactions enforced by the frontend are not reliable
        - E.g., Twitter sending a read status update from the frontend
    - Undocumented or hidden API endpoints are also part of the attack surface
- All API functionality is defined by its endpoints, independently of the frontend
    - This includes transactions and proper authorization decisions

The client is irrelevant for security

The attack surface of an API consists of all accessible endpoints, regardless of how and if they are used by the client

@PhilippeDeRyck

How can we use frontend authorization as a security feature?

Using frontend authorization to detect malicious behavior

- APIs remain responsible for enforcing proper authorization decisions
  - Authorization policies ensure that access to endpoints is allowed
  - Data validation techniques ensure that provided data is valid

- Frontend applications can mimic server-side authorization/validation logic
  - This improves the user experience (E.g., hiding features, quick feedback on input)
  - In this case, the API's authorization and validation policies should never fail
    - If they fail, it is likely that someone is messing around with the application
    - Keep track of such failures to detect malicious users (and take pre-emptive action)

- The ***OWASP AppSensor*** project focuses on such security patterns
  - It defines a framework and methodology to detect incoming attacks
  - AppSensor also provides a set of entry points where malicious behavior can be detected

The client is irrelevant to enforce security

Strict security controls on the client make your API security controls an effective detection mechanism for malicious behavior

A security flaw in Grindr let anyone easily hijack user accounts

Zack Whittaker @zackwhittaker / 10:22 PM GMT+2 • October 2, 2020 Comment

Image Credits: SOPA Images / Getty Images

Grindr, one of the world's largest dating and social networking apps for gay, bi, trans, and queer people, has fixed a security vulnerability that allowed anyone to hijack and take control of any user's account using only their email address.

https://techcrunch.com/2020/10/02/grindr-account-hijack-flaw/

To reset a password, Grindr sends the user an email with a clickable link containing an account password reset token.

Grindr's password reset page was leaking password reset tokens to the browser.

Excessive data exposure

- Many APIs expose too much data to the client
  - Excessive data exposure is ranked #3 in the OWASP API Security Top 10
  - This problem is often "invisible", because the frontend does not use the excessive data
  - Real-world incidents lead to account take-over, location triangulation, …

- A common cause of this problem is the automatic marshalling of objects
  - Often, data objects are directly transformed into JSON by the API controller
  - Data objects often contain sensitive fields which should not be sent in responses
    - E.g., password fields on user objects, admin-only or hidden fields, …

- Avoiding the leaking of data is often quite difficult
  - Marking fields as internal-only is a coarse-grained strategy
  - Exposing data based on the user's permissions requires smart authorization policies

If an API automatically exposes data, does it also automatically accept data?

The body of a legitimate request to update the user's name

```json
{
  "name": "Dr. Phil"
}
```

The API uses a framework that automatically transforms JSON data into domain objects, which are then used to update the persisted data

The Java class of the User object

```java
class User {
  String name;
  String email;
  String password;
}
```

Without filtering the input properties, the API becomes vulnerable to mass assignment

The body of a malicious request to update the user's name

```json
{
  "name": "lol",
  "email": "evil@maliciousfood.com",
  "password": "$2y$13$VeZMDUpYdTvXs7/HB68KPeeetomDafc4huZGE/zr9V4318bWRcDxu"
}
```

Mass assignment

- Many APIs fail to restrict the data fields that a client is allowed to update
  - Mass assignment is ranked #6 in the OWASP API Security Top 10
  - This problem is "invisible", since the frontend never assigns values to these fields
  - Real-world incidents lead to overwriting passwords, updating product prices, …
- A common cause of this problem is the automatic marshalling of objects
  - Incoming JSON data is automatically transformed into internal data objects
  - When JSON fields are not restricted, the JSON can include any field that exists on the object
    - E.g., password fields on user objects, a price field for a webshop product, …
  - Data storage frameworks often use this data to auto-update objects in storage
- Avoiding mass assignment is often not straightforward
  - One strategy is to transform JSON data into a data transfer object (DTO) first
  - Dynamic assignments based on the user's permissions require smart authorization policies

Test your APIs in their natural habitat

Make sure your API behaves the way you think it does. Code analysis is only one aspect. Runtime testing is necessary to get the full picture.

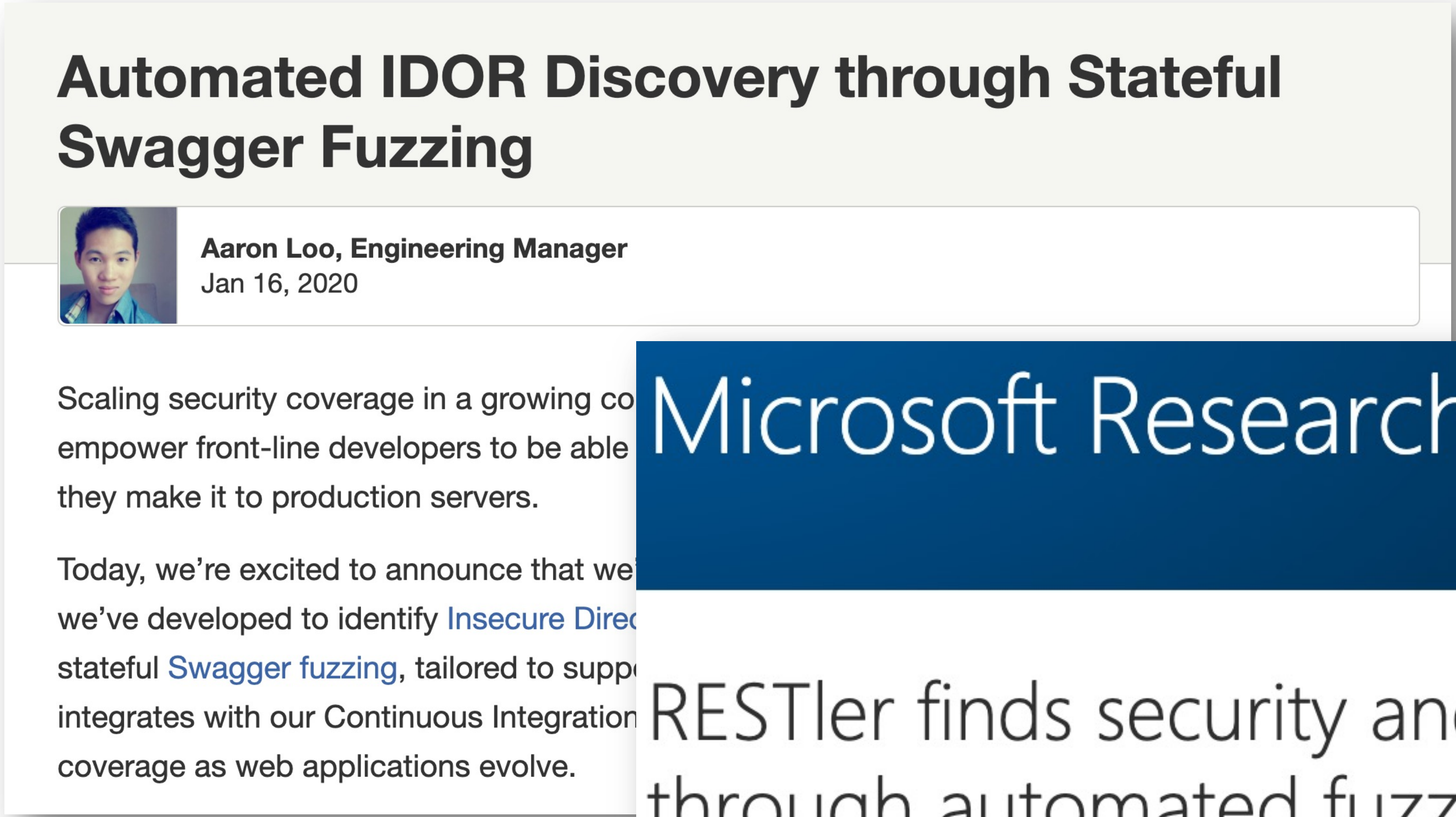
What is the best strategy to avoid data exposure / mass assignment problems?

A Obfuscating the code of the frontend application

B Deploying a Web Application Firewall

C Carefully testing each API endpoint

D Defining an API contract for each endpoint

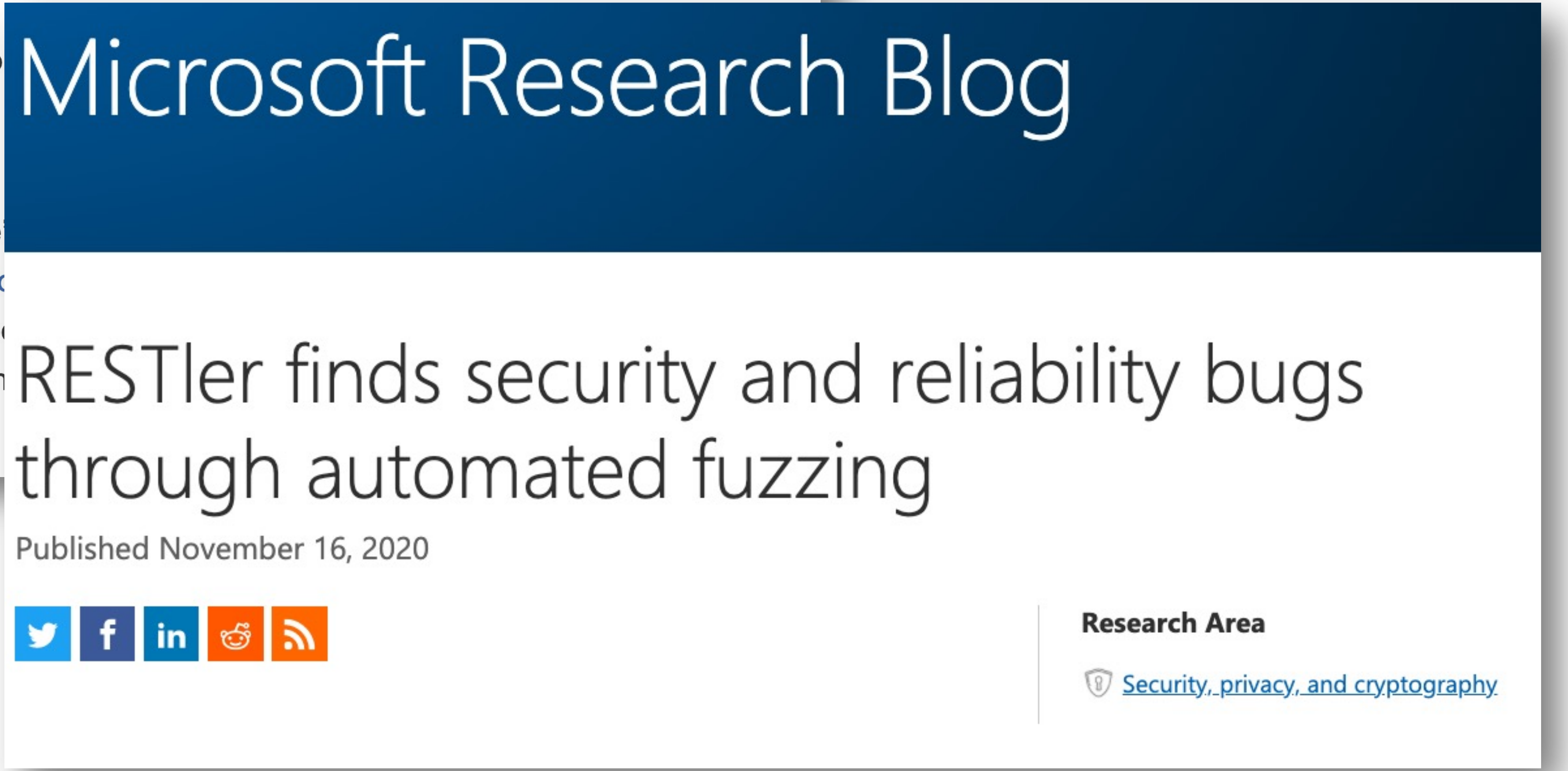


Automated IDOR Discovery through Stateful Swagger Fuzzing

Aaron Loo, Engineering Manager
Jan 16, 2020

Scaling security coverage in a growing co
empower front-line developers to be able
they make it to production servers.

Today, we're excited to announce that we
we've developed to identify Insecure Direc
stateful Swagger fuzzing, tailored to supp
integrates with our Continuous Integration
coverage as web applications evolve.



Microsoft Research Blog

RESTler finds security and reliability bugs through automated fuzzing

Published November 16, 2020

Research Area

Security, privacy, and cryptography

https://engineeringblog.yelp.com/2020/01/automated-idor-discovery-through-stateful-swagger-fuzzing.html
https://www.microsoft.com/en-us/research/blog/restler-finds-security-and-reliability-bugs-through-automated-fuzzing/

An example of a YAML-based OpenAPI contract

```yaml
openapi: 3.0.0
info:
  title: Restograde API
  description: The Restograde API
  version: 0.0.1
servers:
  - url: https://api.restograde.com
    description: The Restograde production API
paths:
  /restaurants:
    get:
      summary: Returns a list of restaurants.
      description: Restaurants are awesome. So are you!
      responses:
        '200': # status code
          description: A JSON array of restaurant names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

AUTOMATED API SECURITY TESTING

- OpenAPI contracts are the de-facto standard for describing modern APIs
  - OpenAPI contracts can be used for contract-first development
  - OpenAPI contracts serve as input for testing and documentation generation
- An OpenAPI contract defines requests and responses for endpoints
  - HTTP methods, content types, request body and response structure
  - Data objects can be defined with re-usable data models
- API security tools can use OpenAPI contracts to determine legitimate traffic
  - Fuzzing / scanning tools use contracts to generate automatic test cases
  - Gateways / firewalls use contracts to determine the nature of legitimate traffic

Use Swagger/OpenAPI definitions for security

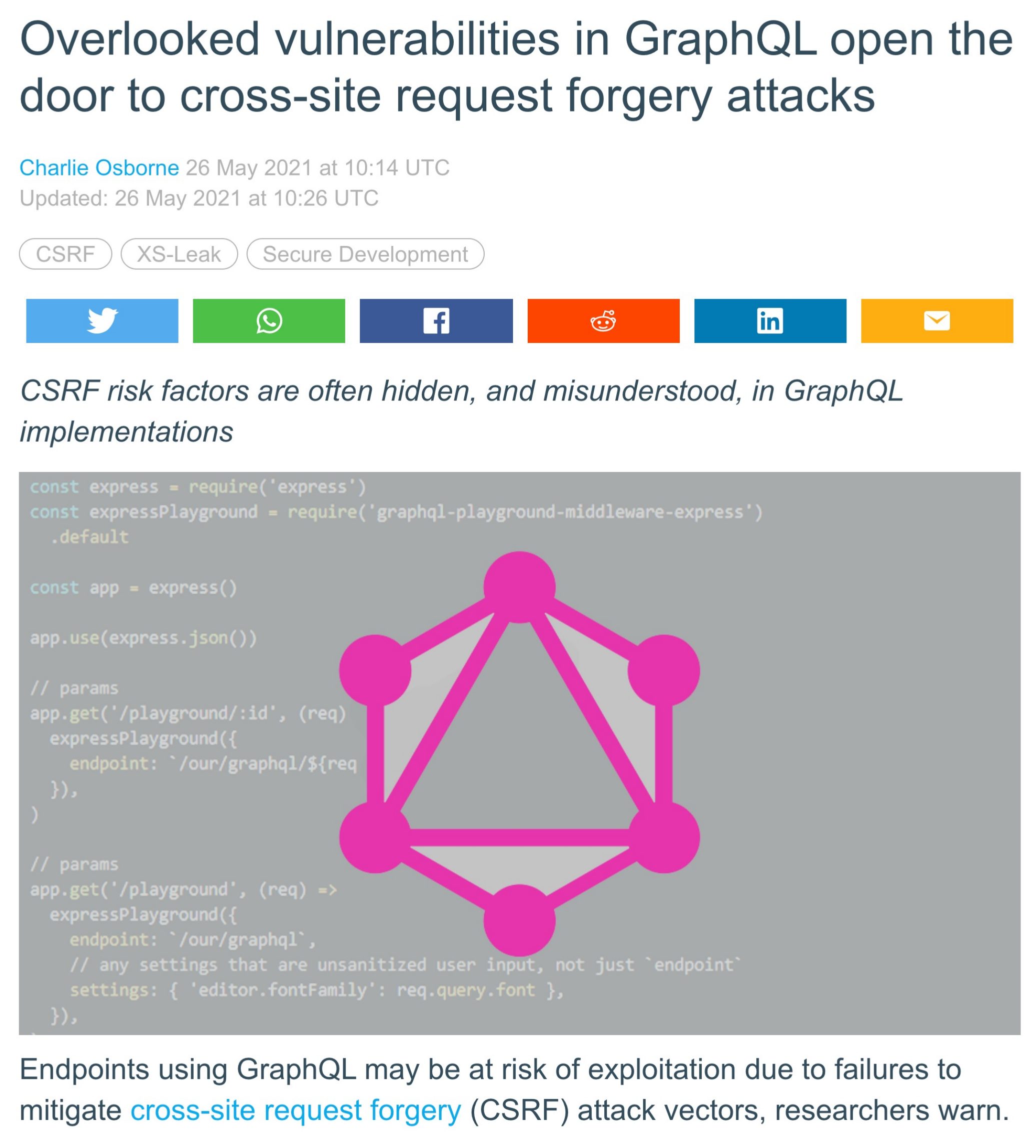
Write Swagger/OpenAPI definitions to specify the behavior of your API. Security tools consume such definitions for automatic detection and protection.

A Python Flask API endpoint

```python
@app.route('/')
def my_first_api_endpoint():
  json_data = json.loads(request.data)
  ...
  return "", 200
```

Which HTTP methods are accepted by this endpoint?

An examination of enterprise endpoints using GraphQL revealed that configuration issues in implementations might be exposing systems to unnecessary risks.



Overlooked vulnerabilities in GraphQL open the door to cross-site request forgery attacks

Charlie Osborne 26 May 2021 at 10:14 UTC
Updated: 26 May 2021 at 10:26 UTC

CSRF XS-Leak Secure Development

CSRF risk factors are often hidden, and misunderstood, in GraphQL implementations

```
const express = require('express')
const expressPlayground = require('graphql-playground-middleware-express')
  .default

const app = express()

app.use(express.json())

// params
app.get('/playground/:id', (req)
  expressPlayground({
    endpoint: `/our/graphql/${req
  }),
)

// params
app.get('/playground', (req) =>
  expressPlayground({
    endpoint: `/our/graphql`,
    // any settings that are unsanitized user input, not just `endpoint`
    settings: { 'editor.fontFamily': req.query.font },
  }),
```

Endpoints using GraphQL may be at risk of exploitation due to failures to mitigate cross-site request forgery (CSRF) attack vectors, researchers warn.

https://portswigger.net/daily-swig/overlooked-vulnerabilities-in-graphql-open-the-door-to-cross-site-request-forgery-attacks

Flask defaults to GET, but supports explicit configuration of allowed HTTP methods

A Python Flask API endpoint

```
@app.route('/', methods=['POST'])
def my_first_api_endpoint():
  json_data = json.loads(request.data)
  ...
  return "", 200
```

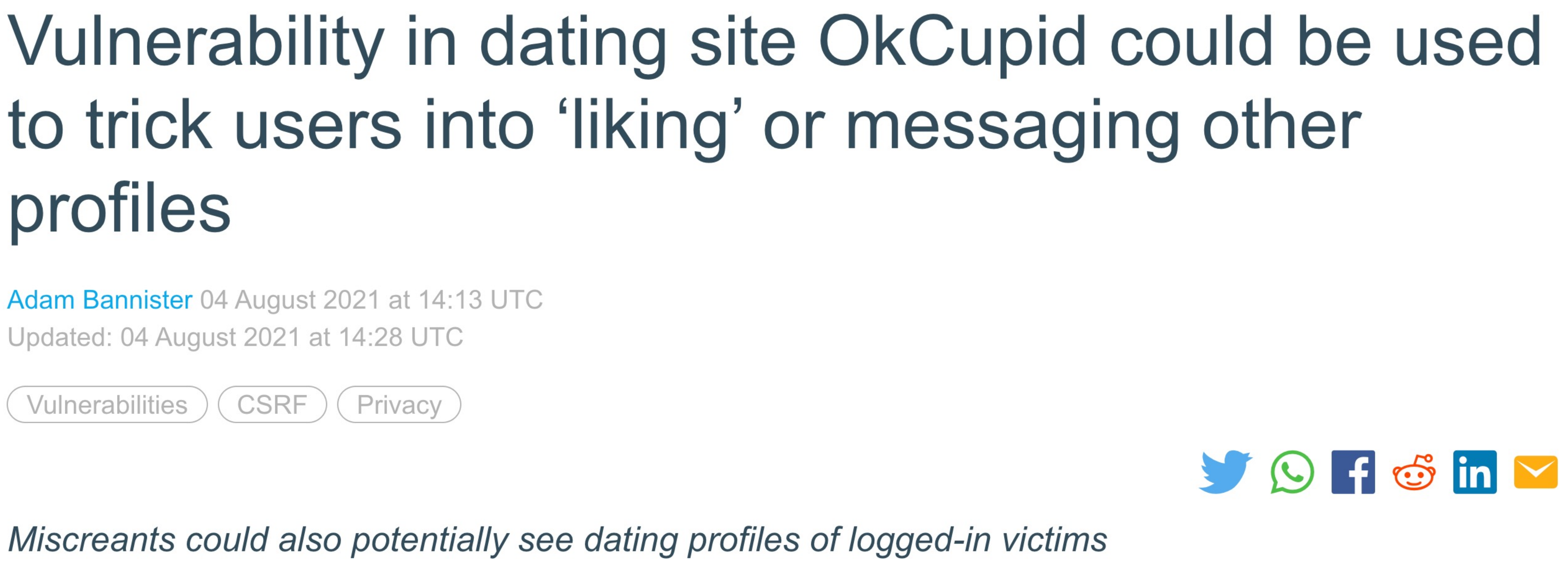
Ensure unexpected HTTP methods are rejected

Every API endpoint should be tested to ensure it only accepts expected HTTP methods and rejects all other methods.

A Python Flask API endpoint

```python
@app.route('/', methods=['POST'])
def my_first_api_endpoint():
  json_data = json.loads(request.data)
  ...
  return "", 200
```

Which HTTP content types are accepted by this endpoint?



Vulnerability in dating site OkCupid could be used to trick users into ‘liking’ or messaging other profiles

Adam Bannister 04 August 2021 at 14:13 UTC
Updated: 04 August 2021 at 14:28 UTC

Vulnerabilities CSRF Privacy

Miscreants could also potentially see dating profiles of logged-in victims

https://portswigger.net/daily-swig/vulnerability-in-dating-site-okcupid-could-be-used-to-trick-users-into-liking-or-messaging-other-profiles

“

Zhu also investigated whether other sites’ authenticated endpoints similarly accepted POSTs with content-type: text/plain, despite expecting JSON.

”

CONTENT TYPE CONFUSION

- Content type confusion can lead to CSRF attacks on JSON endpoints
  - Form fields can be named in such a way that the data becomes valid JSON
  - The form can be defined with a ***text/plain*** content type, which submits raw text data
  - A JSON parser will see the data in the body as valid JSON

- Ensure that the backend rejects unexpected content types
  - A backend allows form-submitted JSON can become vulnerable to CSRF attacks
  - JSON endpoints should only accept ***application/json*** content types

A form that generates valid JSON upon submission

```
<form method="POST" enctype="text/plain">
  <input type="hidden" name='{"title":"' value='...","content": "..."}'>
</form>
```

@PhilippeDeRyck

By default, Flask accepts any content type, including JSON, form-based content types, and "text/plain"

A Python Flask API endpoint

```
@app.route('/', methods=['POST'])
def my_first_api_endpoint():
  json_data = json.loads(request.data)
  ...
  return "", 200
```

Using *request.json* instead of *request.data* only returns a value if the content type is set to "application/json"

Restricting content types in Flask

```python
# Decorator to restrict content types
def content_type(allowed_content_type):
  def decorated(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
      ct = request.headers.get('Content-Type', '')
      if ct.lower() == allowed_content_type.lower():
        return f(*args, **kwargs)

      raise UnsupportedMediaType
    return wrapper
  return decorated

@app.route('/', methods=['POST'])
@content_type('application/json')
def my_first_api_endpoint():
  json_data = request.json
  ...
  return "", 200
```

This endpoint only accepts POST requests with the content type set to "application/json"

Reject unexpected content types

APIs should not be flexible in the way they accept incoming requests. Define the expected content type and reject anything else.

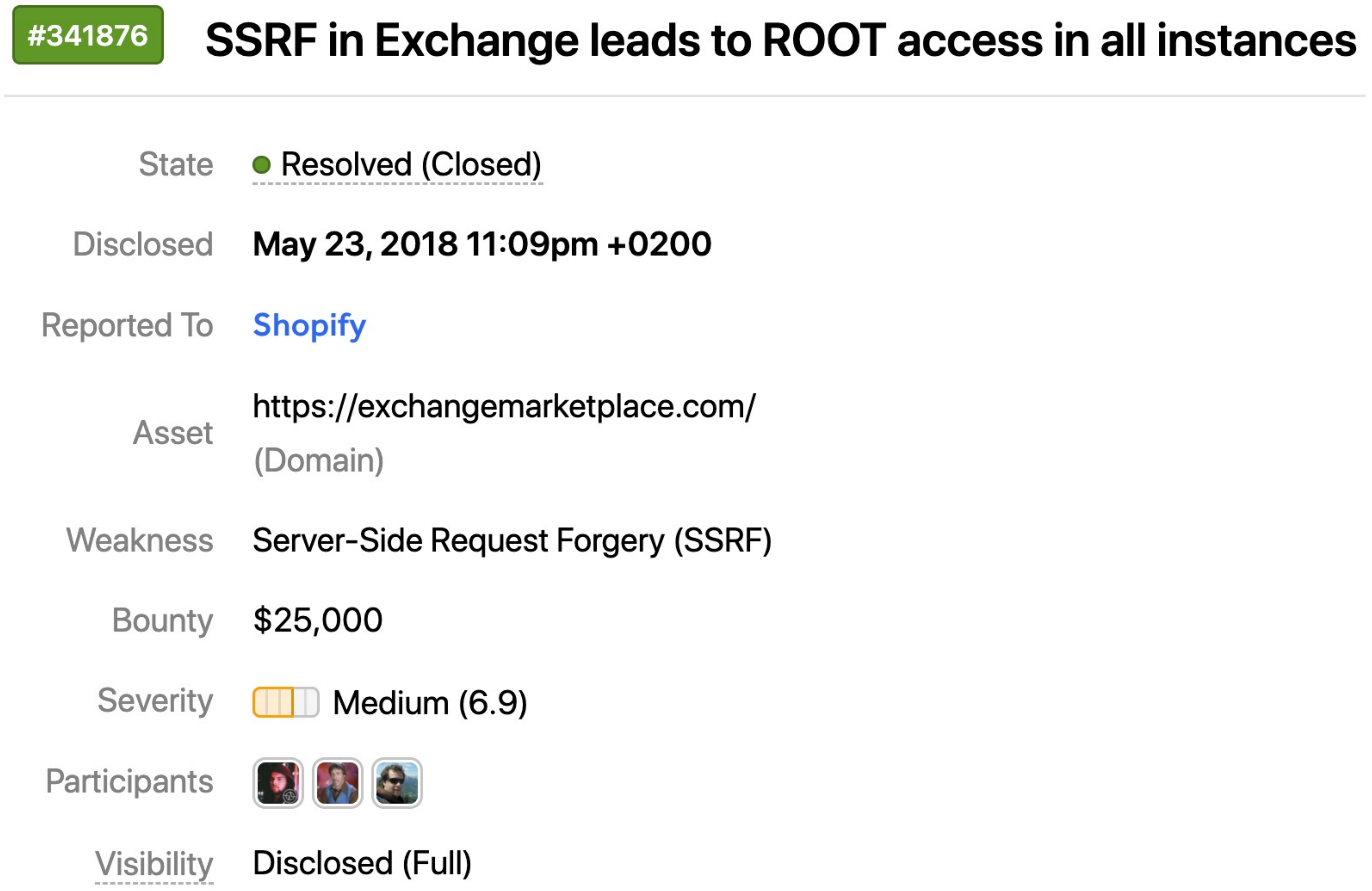
An example of a YAML-based OpenAPI contract

```yaml
openapi: 3.0.0
info:
  title: Restograde API
  description: The Restograde API
  version: 0.0.1
servers:
  - url: https://api.restograde.com
    description: The Restograde production API
paths:
  /restaurants:
    get:
      summary: Returns a list of restaurants.
      description: Restaurants are awesome. So are you!
      responses:
        '200': # status code
          description: A JSON array of restaurant names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

This OpenAPI definition clearly states the expected HTTP method (and content type for POST requests)

Make OpenAPI definitions your new religion

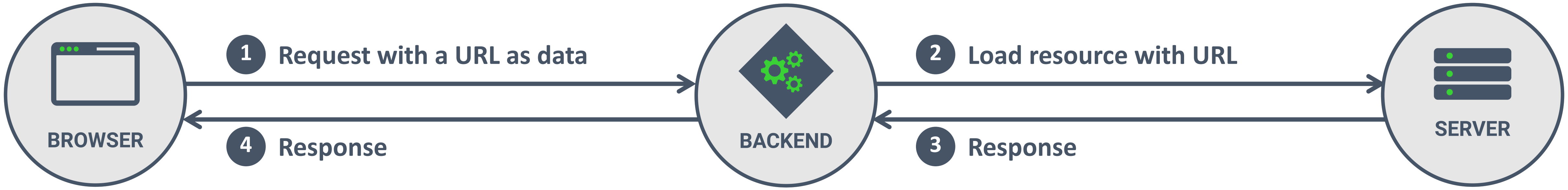
OpenAPI definitions are unambiguous and contain tons of relevant information. Integrate them into your SDLC from the early design phases.



#341876 SSRF in Exchange leads to ROOT access in all instances

State	● Resolved (Closed)
Disclosed	**May 23, 2018 11:09pm +0200**
Reported To	**Shopify**
Asset	https://exchangemarketplace.com/ (Domain)
Weakness	Server-Side Request Forgery (SSRF)
Bounty	$25,000
Severity	Medium (6.9)
Participants	
Visibility	Disclosed (Full)

https://hackerone.com/reports/341876



1 *A request from the browser with a URL as data*

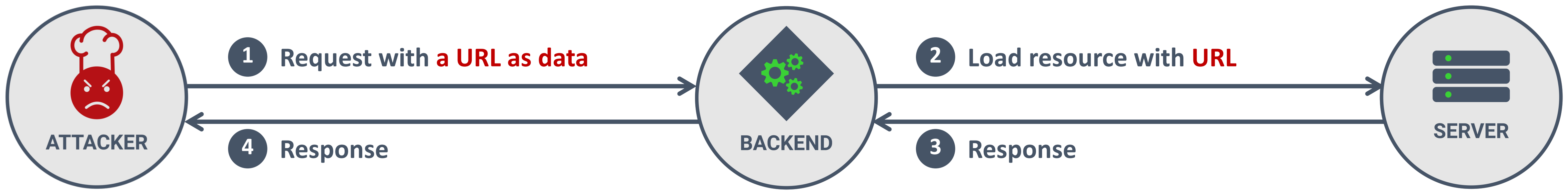
```
POST /restaurants HTTP/1.1
Host: restograde.com

name=My+Restaurant&img=https%3A%2F%2Fimg.example.com%2Frestaurant.png
```

2 *A request from the backend to fetch the image from the provided URL*

```
GET /restaurant.png HTTP/1.1
Host: img.example.com
```

@PhilippeDeRyck



1 *A request from the browser with a URL as data*

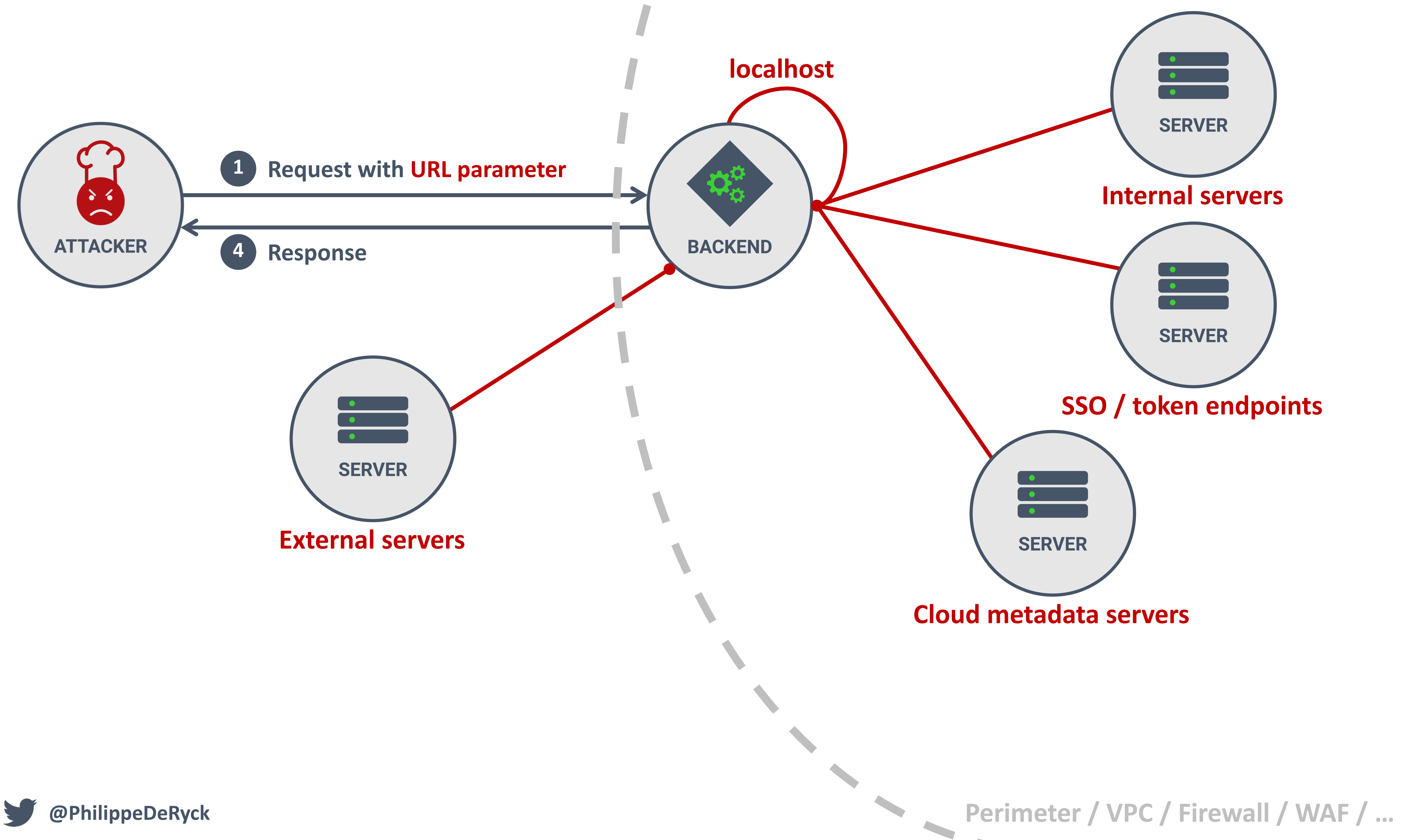
```
POST /restaurants HTTP/1.1
Host: restograde.com

name=My+Restaurant&img=any endpoint
```

2 *A request from the backend to an arbitrary attacker-provided endpoint*

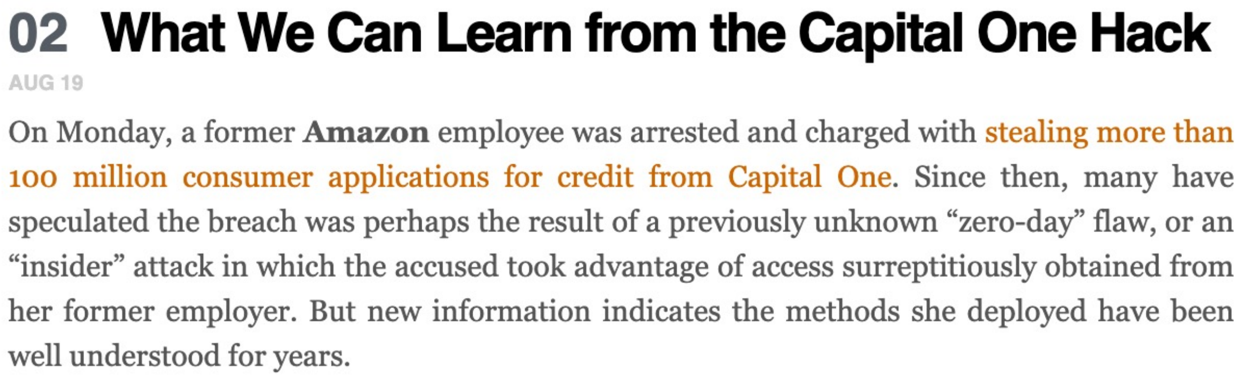
```
GET /anyEndpoint HTTP/1.1
Host: anyserver.example.com
```

@PhilippeDeRyck



ATTACKER
1 Request with URL parameter
4 Response
BACKEND
localhost
SERVER
Internal servers
SERVER
SSO / token endpoints
SERVER
Cloud metadata servers
SERVER
External servers
Perimeter / VPC / Firewall / WAF / ...

“The type of vulnerability exploited by the intruder in the Capital One hack is a well-known method called a “Server Side Request Forgery” (SSRF) attack, in which a server (in this case, CapOne’s WAF) can be tricked into running commands that it should never have been permitted to run, including those that allow it to talk to the metadata service.”



02 What We Can Learn from the Capital One Hack

AUG 19

On Monday, a former **Amazon** employee was arrested and charged with stealing more than 100 million consumer applications for credit from Capital One. Since then, many have speculated the breach was perhaps the result of a previously unknown “zero-day” flaw, or an “insider” attack in which the accused took advantage of access surreptitiously obtained from her former employer. But new information indicates the methods she deployed have been well understood for years.

https://krebsonsecurity.com/2019/08/what-we-can-learn-from-the-capital-one-hack/

Server-Side Request Forgery (SSRF)

- The attacker controls a URL, tricking a server into making a request
  - SSRF typically results in a GET request being issued, but POST requests also occur
  - The attacker can provide an arbitrary target URL, including parameters
    - E.g., loading images, calling internal systems, executing webhooks, …
- SSRF executes within the application's perimeter, increasing its potential
  - Publicly unreachable services become reachable
  - Requests can include authentication information when added automatically
    - E.g., by token middleware or mTLS configuration settings
- SSRF is a server-side variation of Cross-Site Request Forgery (CSRF)
  - With CSRF, the request is launched from a user's browser
  - CSRF attacks targeting employees or admin also gave the attacker internal access

So how can we restrict destinations of server-side requests?

Which of these mechanisms are suitable to ensure an IP is not pointing to localhost?

- A String-based matching
- B A regular expression
- C A custom validation function
- D A carefully-selected IP address validation library

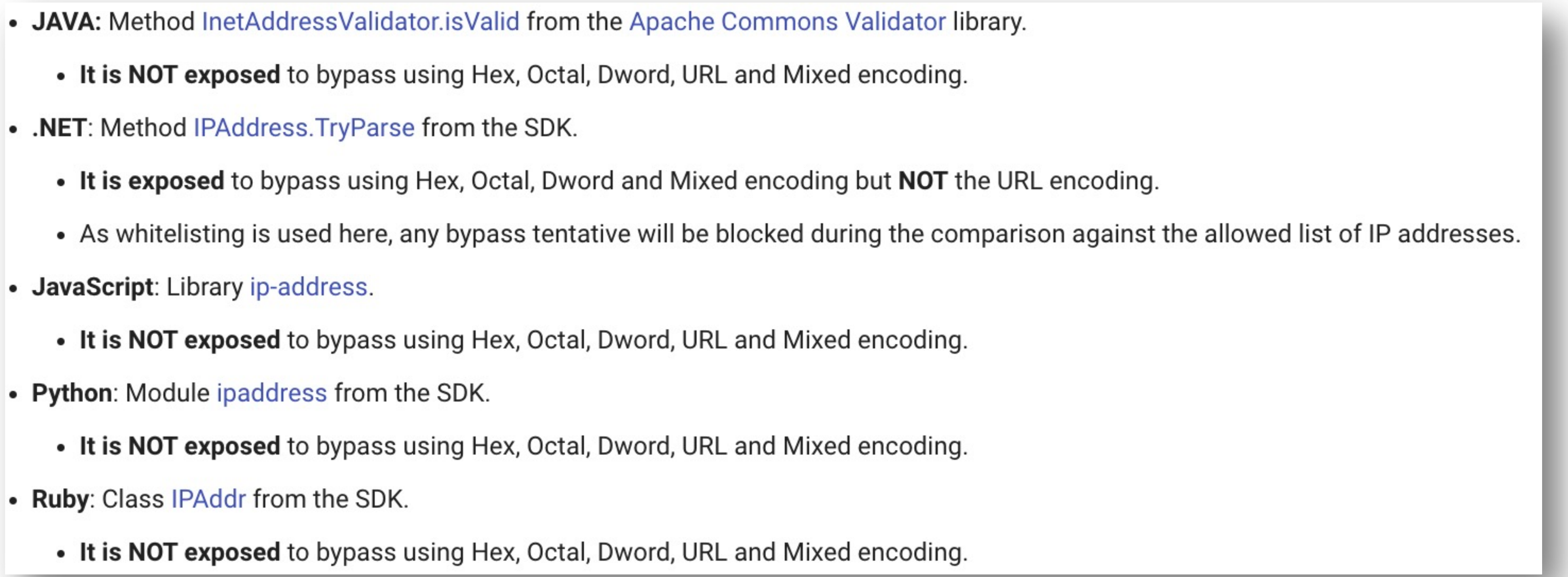
Normal IPv4 address(dotted decimal):	`127.0.0.1`
0-optimized dotted decimal:	`127.1`
0-optimized dotted decimal:	`127.0.0000000000000000000000000000000001`
Octal:	`0177.0.0.01`
Octal:	`00000000177.000.0.00000001`
Octal:	`0177.0.0.0000001`
Octal:	`0000177.000000000000000000.00000000000.00000000001`
Hexadecimal:	`0x7f.0x0.0x0.0x1`
Hexadecimal:	`0x7f000001`
Hexadecimal:	`0xDEADBEEF7f000001`
Hexadecimal:	`0xBADF00D7f000001`
Hexadecimal:	`0xBAAAaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa7f000001`
Dword (non-dotted decimal):	`2130706433`
Binary:	`01111111000000000000000000000001`

Mixed:	`00000000000000000000000000000000000000000000000000177.1`
Mixed:	`0x7f.1`
Mixed:	`127.0x1`
IPv6:	`0000000000000:0000:0000:0000:0000:00000000000000:0000:1`
IPv6:	`0000:0000:0000:0000:0000:0000:0000:0001`
IPv6:	`0:0:0:0:0:0:0:1`
IPv6:	`0:0:0:0::0:0:1`
URL-encoded:	`http://%31%32%37%2E%30%2E%30%2E%31`
URL-encoded:	`http://[%3A%3A%31]`

The last two require a mechanism that will URL-decode the IP addresses at input time

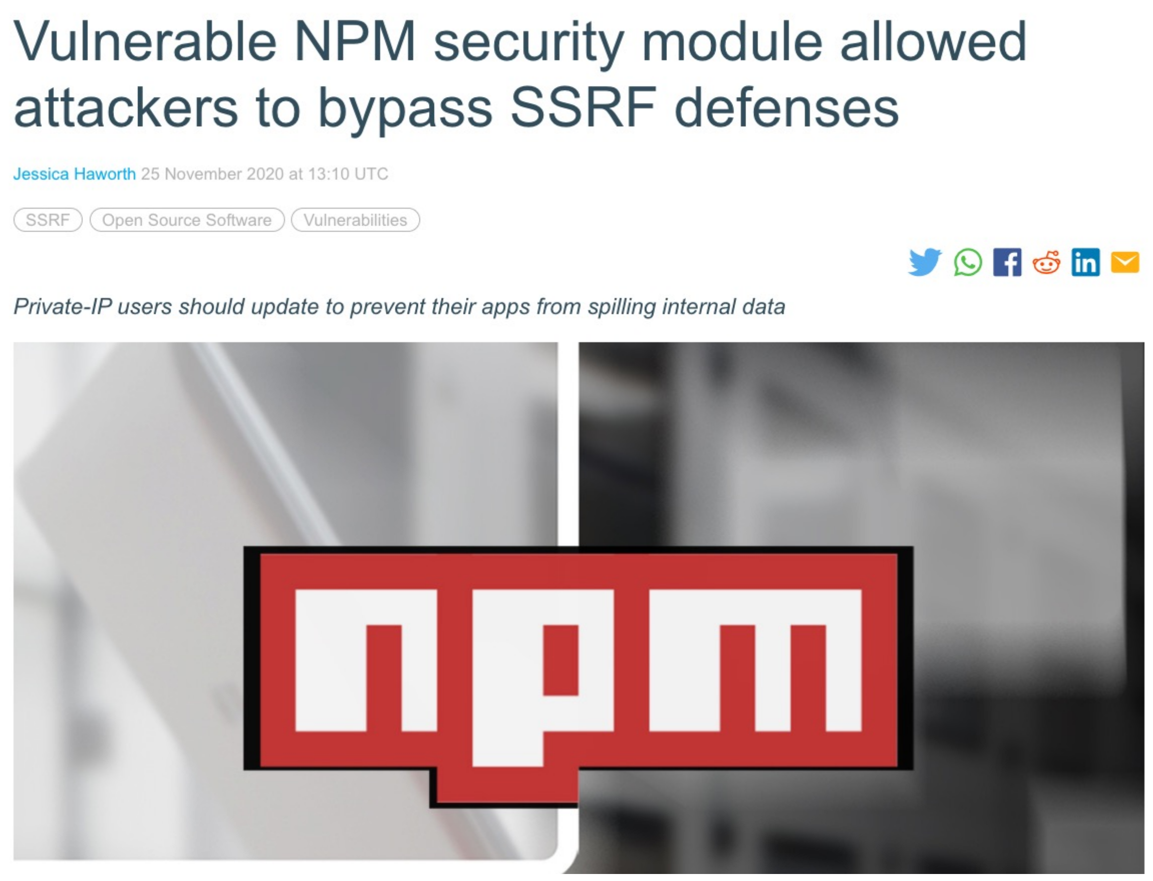
Restricting IP addresses

- IP address validation cannot be done with regexes or custom code
  - It is extremely likely that bypasses against such mechanisms exist
  - Highly recommended to use a solid IP address library
  - Process the input with the library and validate the normalized output of the library



- **JAVA:** Method InetAddressValidator.isValid from the Apache Commons Validator library.
  - **It is NOT exposed** to bypass using Hex, Octal, Dword, URL and Mixed encoding.
- **.NET**: Method IPAddress.TryParse from the SDK.
  - **It is exposed** to bypass using Hex, Octal, Dword and Mixed encoding but **NOT** the URL encoding.
  - As whitelisting is used here, any bypass tentative will be blocked during the comparison against the allowed list of IP addresses.
- **JavaScript**: Library ip-address.
  - **It is NOT exposed** to bypass using Hex, Octal, Dword, URL and Mixed encoding.
- **Python**: Module ipaddress from the SDK.
  - **It is NOT exposed** to bypass using Hex, Octal, Dword, URL and Mixed encoding.
- **Ruby**: Class IPAddr from the SDK.
  - **It is NOT exposed** to bypass using Hex, Octal, Dword, URL and Mixed encoding.

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



Vulnerable NPM security module allowed attackers to bypass SSRF defenses

Jessica Haworth 25 November 2020 at 13:10 UTC

SSRF | Open Source Software | Vulnerabilities

Private-IP users should update to prevent their apps from spilling internal data

https://portswigger.net/daily-swig/vulnerable-npm
-security-module-allowed-attackers-to-bypass-ssrf-defenses

“

The code logic was utilizing simple Regular Expression, therefore not accounting for variations of localhost, and other private-ip ranges, as predicted,” the researchers explained.

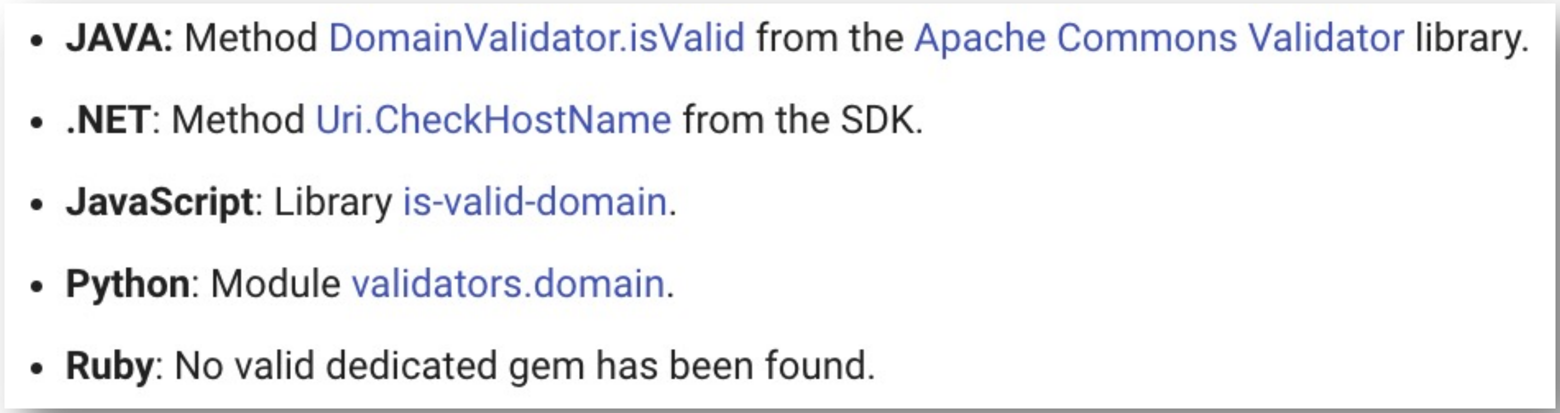
”

Use a solid IP address library

Inspect your IP address library to ensure it properly handles IP address validation. Aim to normalize addresses before analyzing them.

DEALING WITH DOMAINS

- Domains are a bit more straightforward, but involve the use of DNS
  - Validate input using a proper domain validation library
- An attacker can setup their own DNS to resolve domains to internal IPs
  - **Resolve domains to an IP address to validate the final destination**



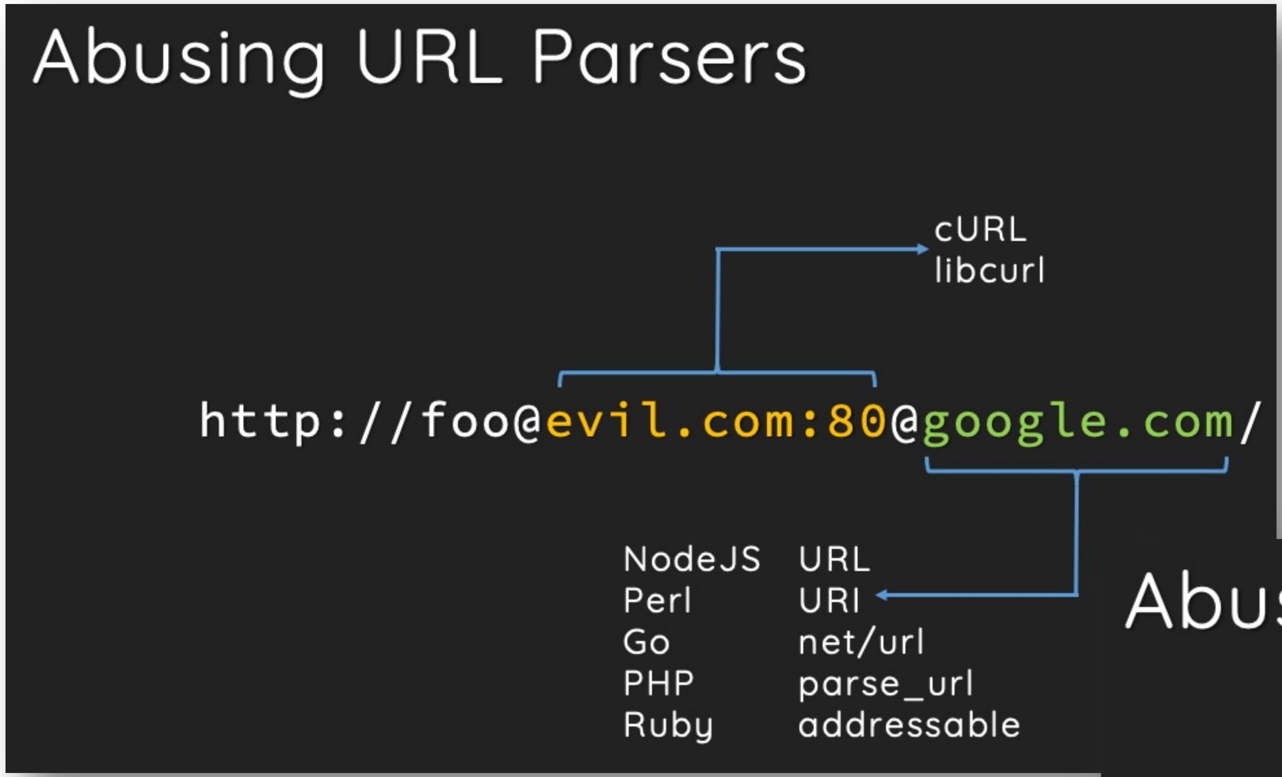
- **JAVA:** Method DomainValidator.isValid from the Apache Commons Validator library.
- **.NET**: Method Uri.CheckHostName from the SDK.
- **JavaScript**: Library is-valid-domain.
- **Python**: Module validators.domain.
- **Ruby**: No valid dedicated gem has been found.

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

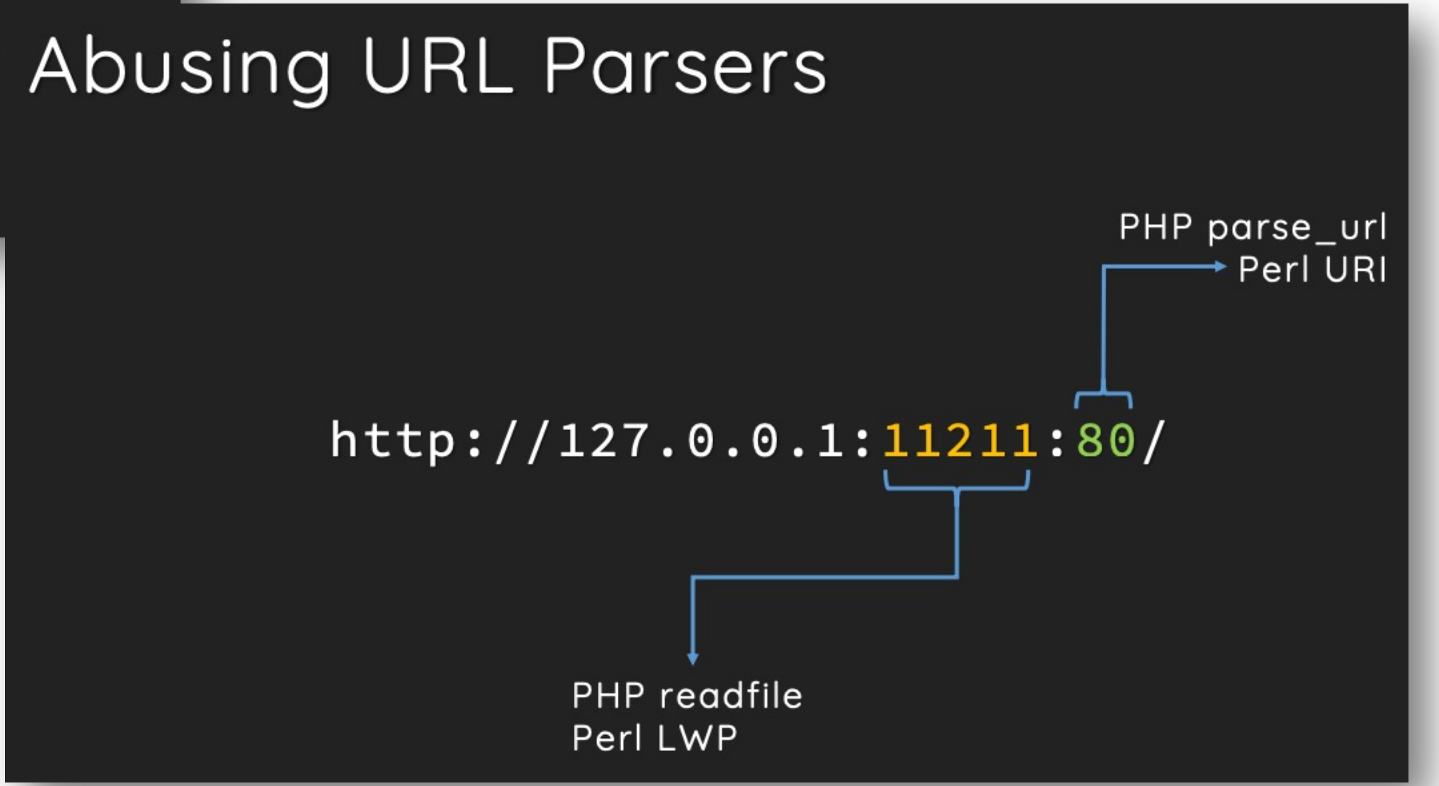
Domains are fine, but how do you handle a URL in a callback or webhook?

Accepting URLs

- Use a URL parsing library to parse the URL
  - Validate the result to ensure it matches what you expect

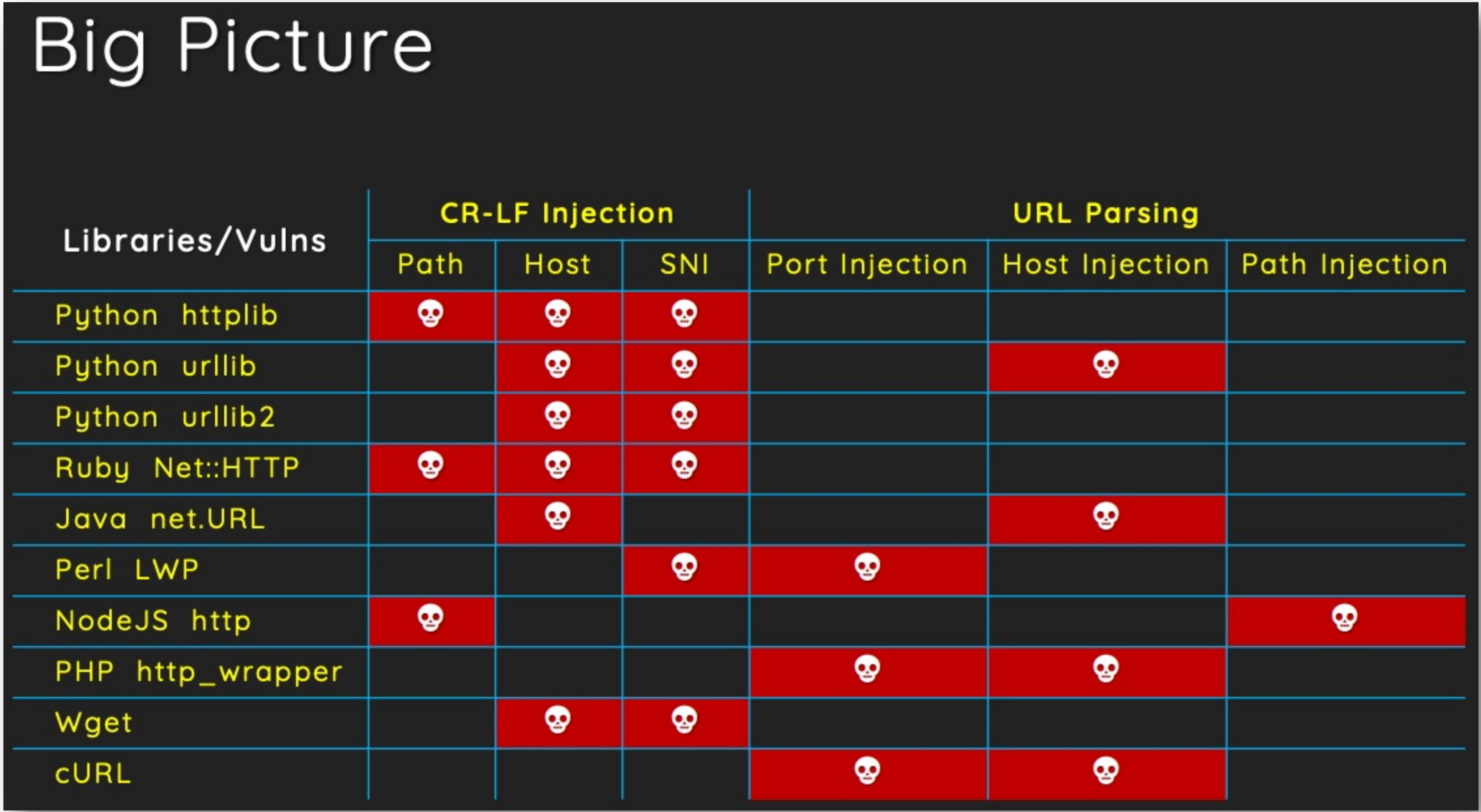


Abusing URL Parsers



Abusing URL Parsers

https://cheatsheetseries.owasp.org/assets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet_Orange_Tsai_Talk.pdf



Big Picture

Libraries/Vulns	CR-LF Injection			URL Parsing		
	Path	Host	SNI	Port Injection	Host Injection	Path Injection
Python httplib	💀	💀	💀			
Python urllib		💀	💀		💀	
Python urllib2		💀	💀			
Ruby Net::HTTP	💀	💀	💀			
Java net.URL		💀			💀	
Perl LWP			💀	💀		
NodeJS http	💀					💀
PHP http_wrapper				💀	💀	
Wget		💀	💀			
cURL				💀	💀	

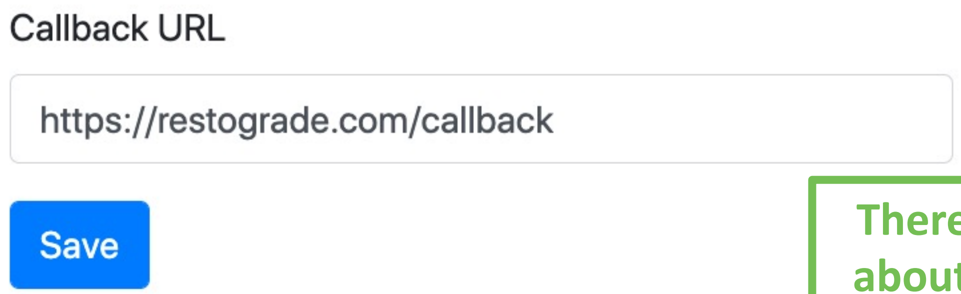
https://cheatsheetseries.owasp.org/assets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet_Orange_Tsai_Talk.pdf

Accepting URLs

- ~~Use a URL parsing library to parse the URL~~
  - ~~Validate the result to ensure it matches what you expect~~

- Try to avoid accepting URLs from the user as input
  - When only allowing a select number of URLs/hosts, allow ID-based selection from a list
  - Fall back on using domains or IP addresses instead of full URLs

- When accepting a URL is unavoidable, accept as little information as possible
  - E.g., force ***https://*** instead of rejecting ***file://***, ***phar://***, ***gopher://***, ***data://***, ***dict://***, …
  - To avoid weird side-effects, it is recommended to accept input in pieces
    - E.g., make the client submit URL data in parts (scheme, host, path, parameters, …)
    - Validate each piece as strict as possible (e.g., reject # or ? in the host part)
  - By leveraging the browser's URL parser, the UX with a single URL can be preserved



Callback URL

https://restograde.com/callback

There is no confusion about the meaning of the data anymore

The code handling the URL input

```
function saveUrl() {
  let strUrl = document.getElementById("cb").value;
  let url = new URL(strUrl);

  let urlData = {
    "scheme": url.protocol,
    "hostname": url.hostname,
    "port": url.port,
    "path": url.pathname,
    "params": url.search,
    "fragment": url.hash
  }

  // Send this data to the backend for processing
}
```

The data received by the API

```
{
  "scheme":"https:",
  "hostname":"restograde.com",
  "port":"",
  "path":"/callback",
  "params":"",
  "fragment":""
}
```

The browser's URL parser is used to parse the URL into components

Accepting URLs

- Try to avoid accepting URLs from the user as input
  - When only allowing a select number of URLs/hosts, allow ID-based selection from a list
  - Fall back on using domains or IP addresses instead of full URLs

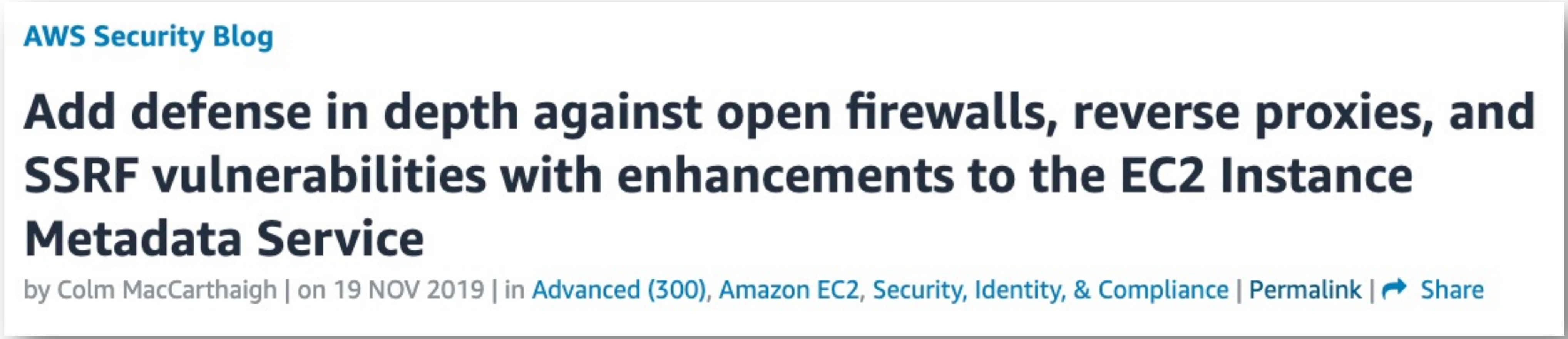
- When accepting a URL is unavoidable, accept as little information as possible
  - E.g., force ***https://*** instead of rejecting ***file://***, ***phar://***, ***gopher://***, ***data://***, ***dict://***, …
  - To avoid weird side-effects, it is recommended to accept input in pieces
    - E.g., make the client submit URL data in parts (scheme, host, path, parameters, …)
    - Validate each piece as strict as possible (e.g., reject # or ? in the host part)
  - By leveraging the browser's URL parser, the UX with a single URL can be preserved

- When there is truly no other option, use a URL parsing library to parse the URL
  - Carefully validate the result to ensure it matches what you expect

Defense-in-depth against SSRF

- Isolate services generating outgoing requests from the main application
  - Network segmentation can help prevent accidental access to internal systems

- Setup proper inter-service authentication to avoid unauthorized requests
  - Simple mechanisms rely on API keys or mutual TLS
  - More complex mechanisms can involve OAuth 2.0 or custom security measures



AWS Security Blog

Add defense in depth against open firewalls, reverse proxies, and SSRF vulnerabilities with enhancements to the EC2 Instance Metadata Service

by Colm MacCarthaigh | on 19 NOV 2019 | in Advanced (300), Amazon EC2, Security, Identity, & Compliance | Permalink | Share

https://aws.amazon.com/blogs/security/defense-in-depth-open-firewalls-reverse-proxies-ssrf-vulnerabilities-ec2-instance-metadata-service/

Do not underestimate the prevalence of SSRF

Aim to remove as much ambiguity as possible by accepting well-defined input and sending requests from isolated hosts with limited permissions.

Key takeaways

1. Make expected behavior as explicit as possible

2. Integrate the use of OpenAPI definitions into your SDLC

3. Use compartmentalization to reduce the impact of vulnerabilities

Thank you for watching!

Connect on social media for more in-depth security content

@PhilippeDeRyck

/in/PhilippeDeRyck