SEVEN THINGS ABOUT API SECURITY

DR. PHILIPPE DE RYCK

https://PragmaticWebSecurity.com

1 Broken object level authorization

2 Broken authentication

3 Broken object property-level authorization

4 Unrestricted resource consumption

5 Broken function level authorization

6 Unrestricted access to sensitive business flows

7 Server-side request forgery

8 Security misconfiguration

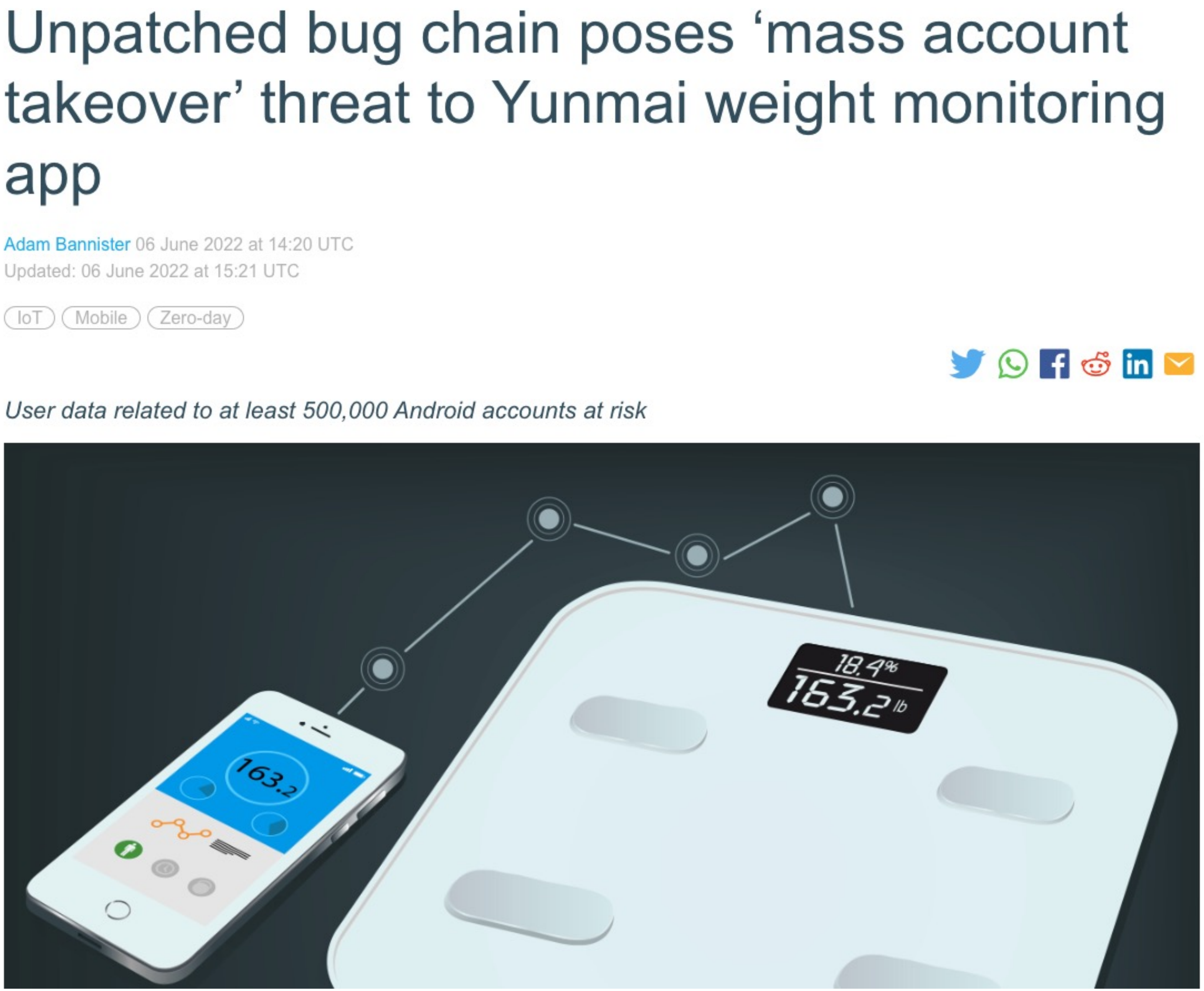
9 Improper inventory management

10 Unsafe consumption of APIs

OWASP

API Security

TOP10



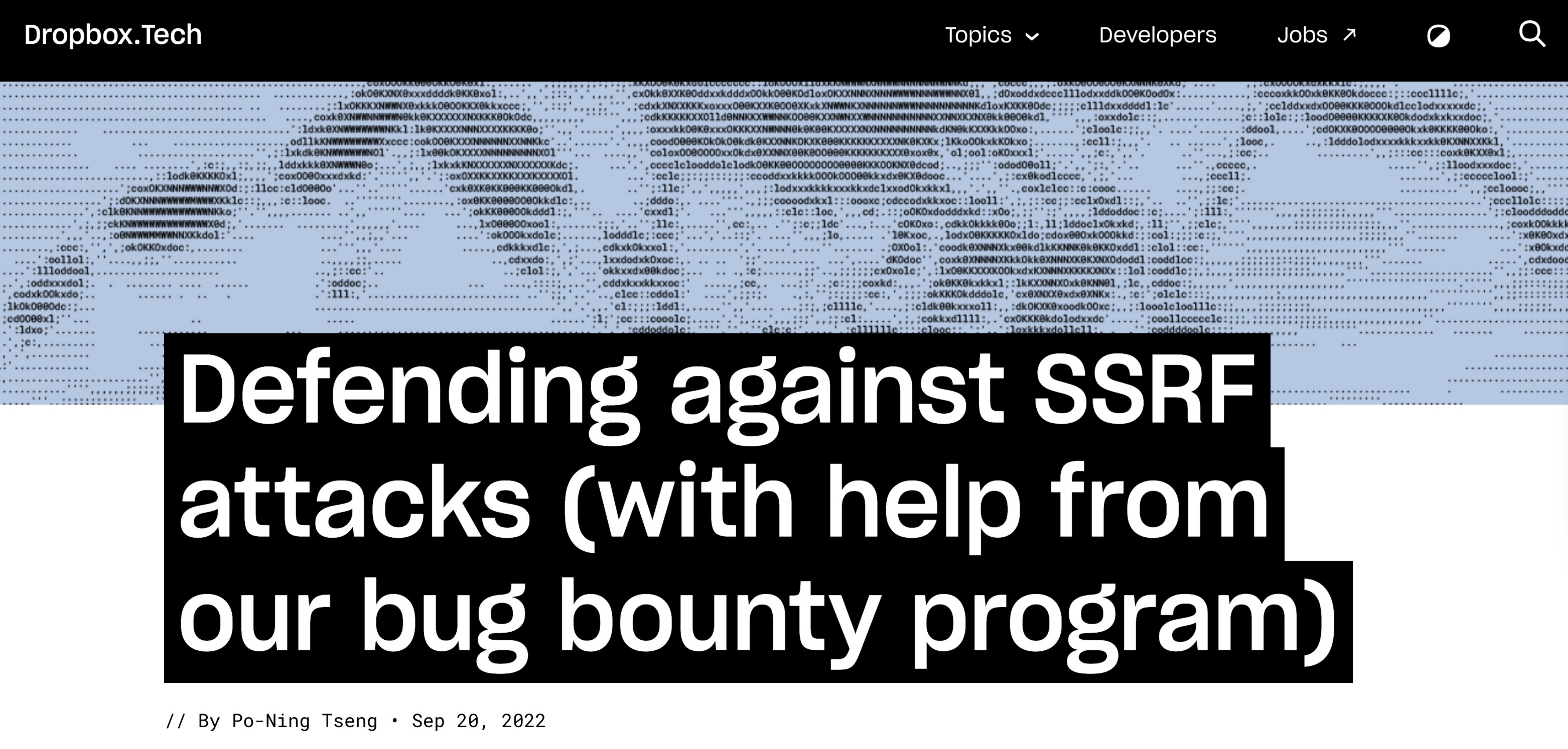
Unpatched bug chain poses 'mass account takeover' threat to Yunmai weight monitoring app

Adam Bannister 06 June 2022 at 14:20 UTC
Updated: 06 June 2022 at 15:21 UTC

IoT Mobile Zero-day

User data related to at least 500,000 Android accounts at risk

https://portswigger.net/daily-swig/unpatched-bug-chain-poses-mass-account-takeover-threat-to-yunmai-weight-monitoring-app



Defending against SSRF attacks (with help from our bug bounty program)

// By Po-Ning Tseng • Sep 20, 2022

https://dropbox.tech/security/bug-bounty-program-ssrf-attack

I am *Dr. Philippe De Ryck*

Pragmatic Web Security
Security for developers

Founder of Pragmatic Web Security

Google Developers Experts

Google Developer Expert

SecAppDev

SecAppDev organizer

I help developers with security

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

https://pdr.online

Grab a copy of the slides ...

https://pragmaticwebsecurity.com/talks

/in/PhilippeDeRyck

@philippederyck.bsky.social

Website icons created by Uniconlabs - Flaticon

1	Broken object level authorization
2	Broken authentication
3	Broken object property-level authorization
4	Unrestricted resource consumption
5	Broken function level authorization
6	Unrestricted access to sensitive business flows
7	Server-side request forgery
8	Security misconfiguration
9	Improper inventory management
10	Unsafe consumption of APIs

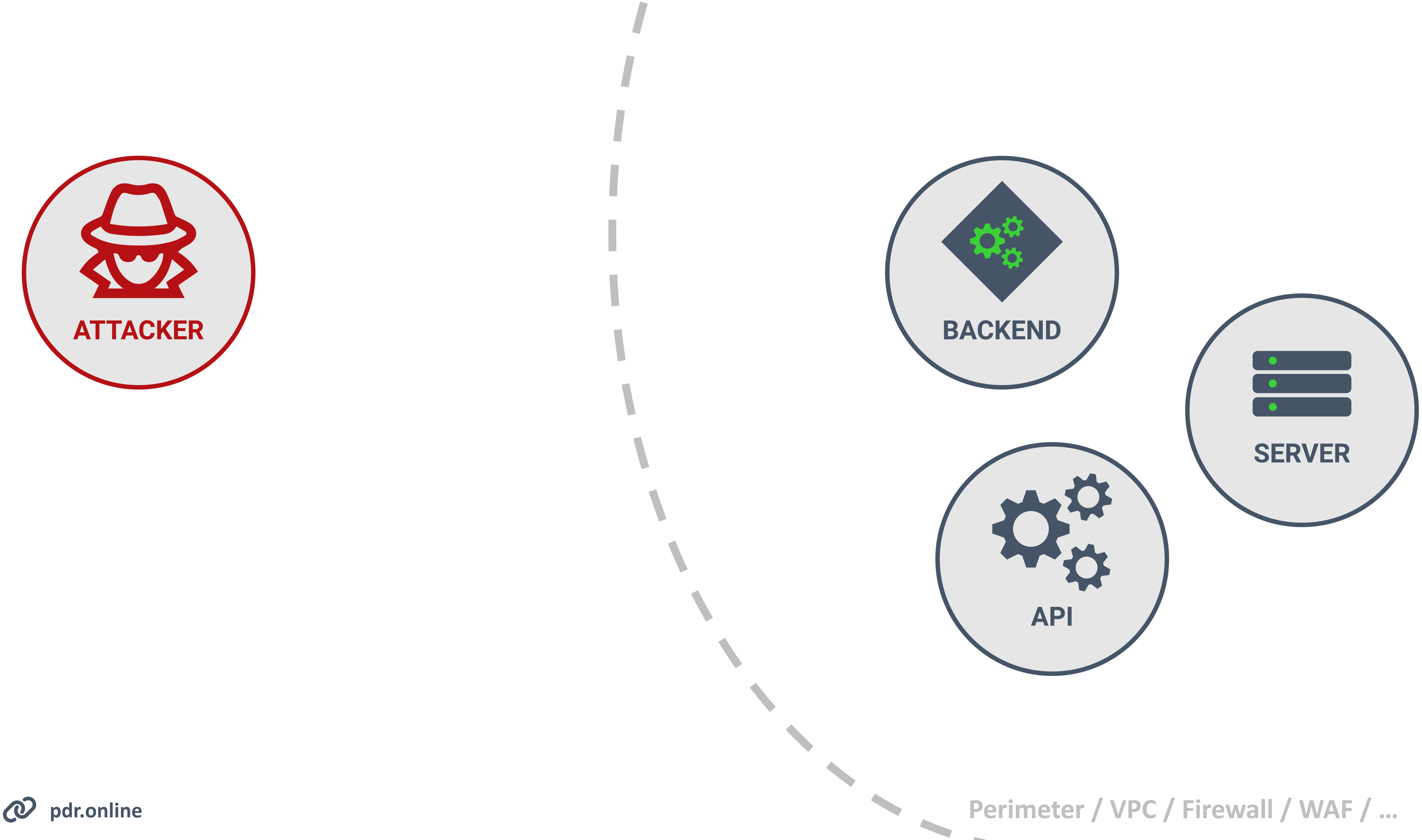
OWASP

API Security

TOP10

Perimeter / VPC / Firewall / WAF / …

Photo by Robert V. Ruggiero on Unsplash
Photo by Thula Na on Unsplash



ATTACKER

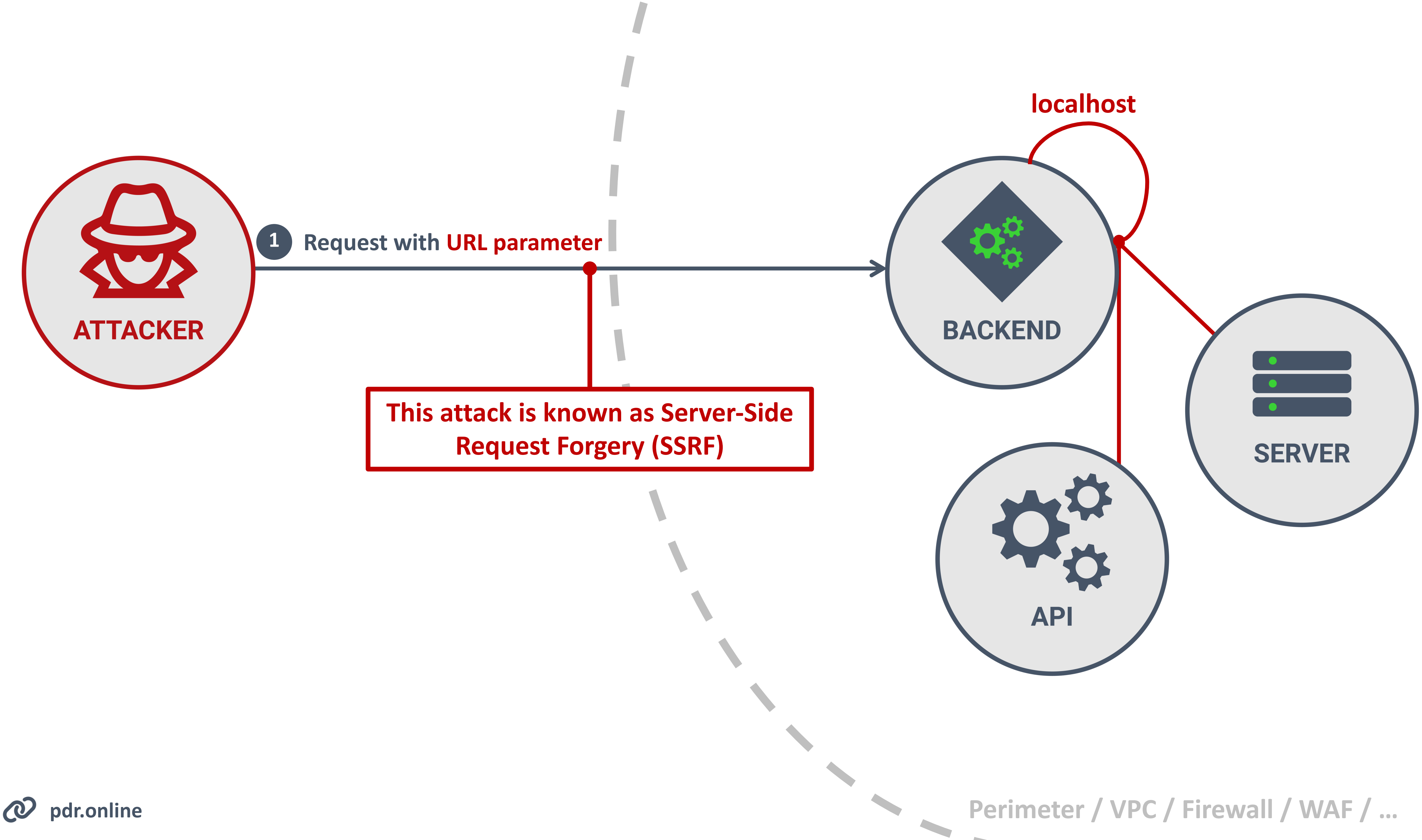
BACKEND

SERVER

API

Perimeter / VPC / Firewall / WAF / ...

Photo by Shalone Cason on Unsplash



ATTACKER

1 Request with URL parameter

This attack is known as Server-Side Request Forgery (SSRF)

localhost

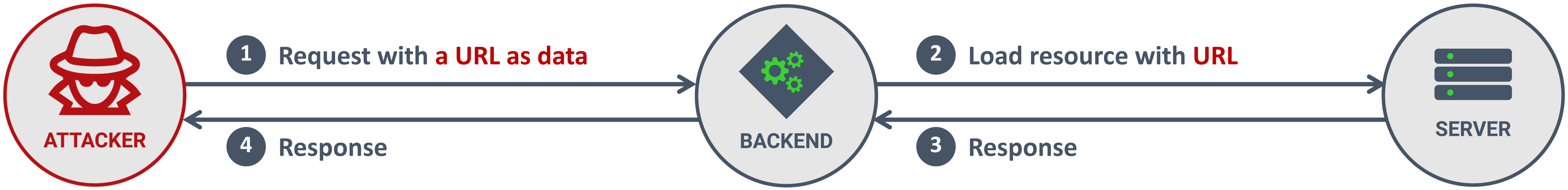
BACKEND

SERVER

API

Perimeter / VPC / Firewall / WAF / ...

How does SSRF happen?



1 Request with **a URL as data**

2 Load resource with **URL**

3 Response

4 Response

SSRF occurs in all kinds of services, such as image loading, link previews, webhooks, proxies, ...

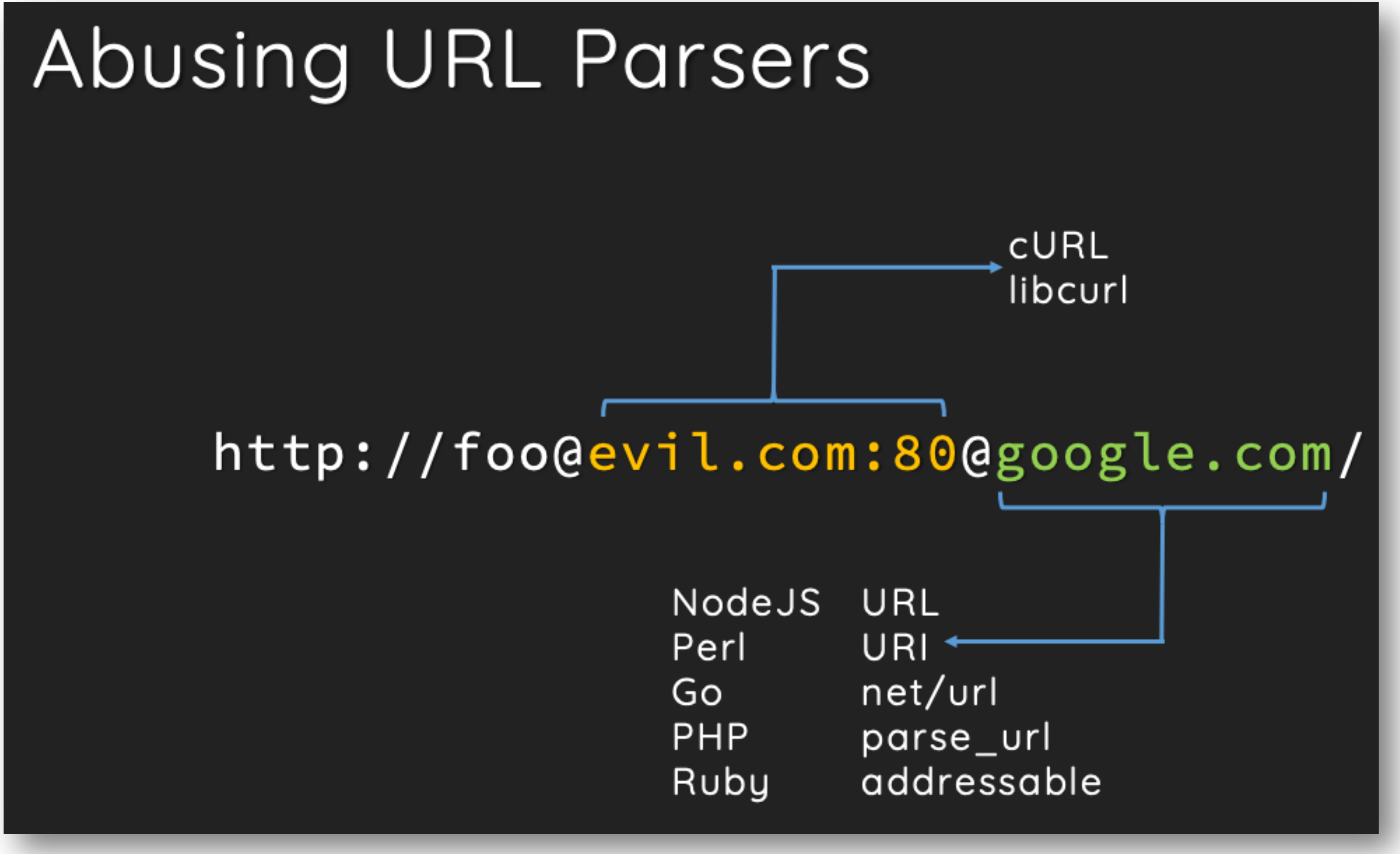
An SSRF vulnerability can lead to requests being sent to localhost, internal hosts, token services, or cloud security services ...

`https://mycdn.example.com/image.png`

Valid input

`https://127.0.0.1:8080`

Invalid input



Abusing URL Parsers

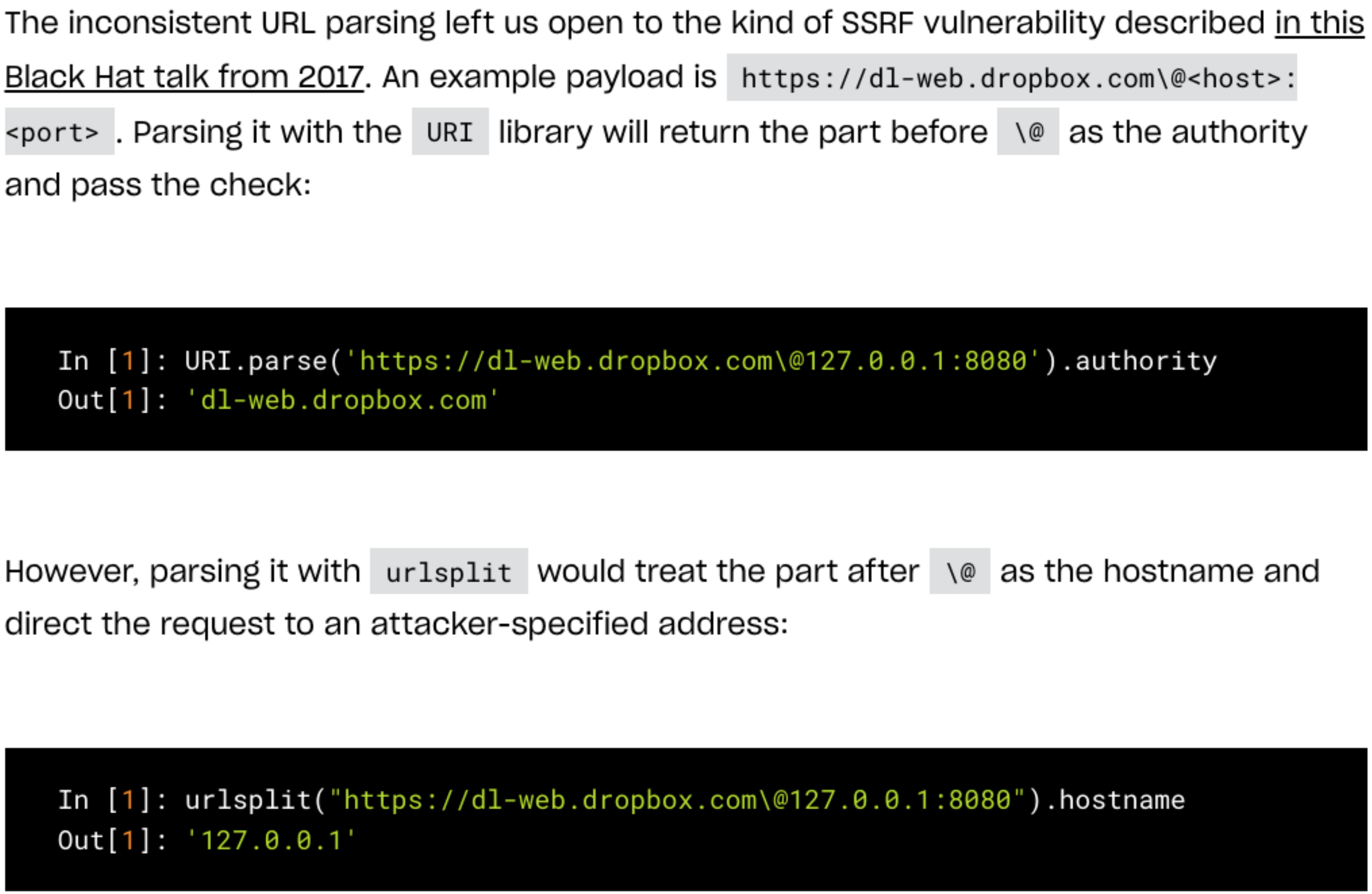
cURL
libcurl

```
http://foo@evil.com:80@google.com/
```

NodeJS	URL
Perl	URI
Go	net/url
PHP	parse_url
Ruby	addressable

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

SSRF at Dropbox



The inconsistent URL parsing left us open to the kind of SSRF vulnerability described in this Black Hat talk from 2017. An example payload is `https://dl-web.dropbox.com\@<host>:<port>`. Parsing it with the `URI` library will return the part before `\@` as the authority and pass the check:

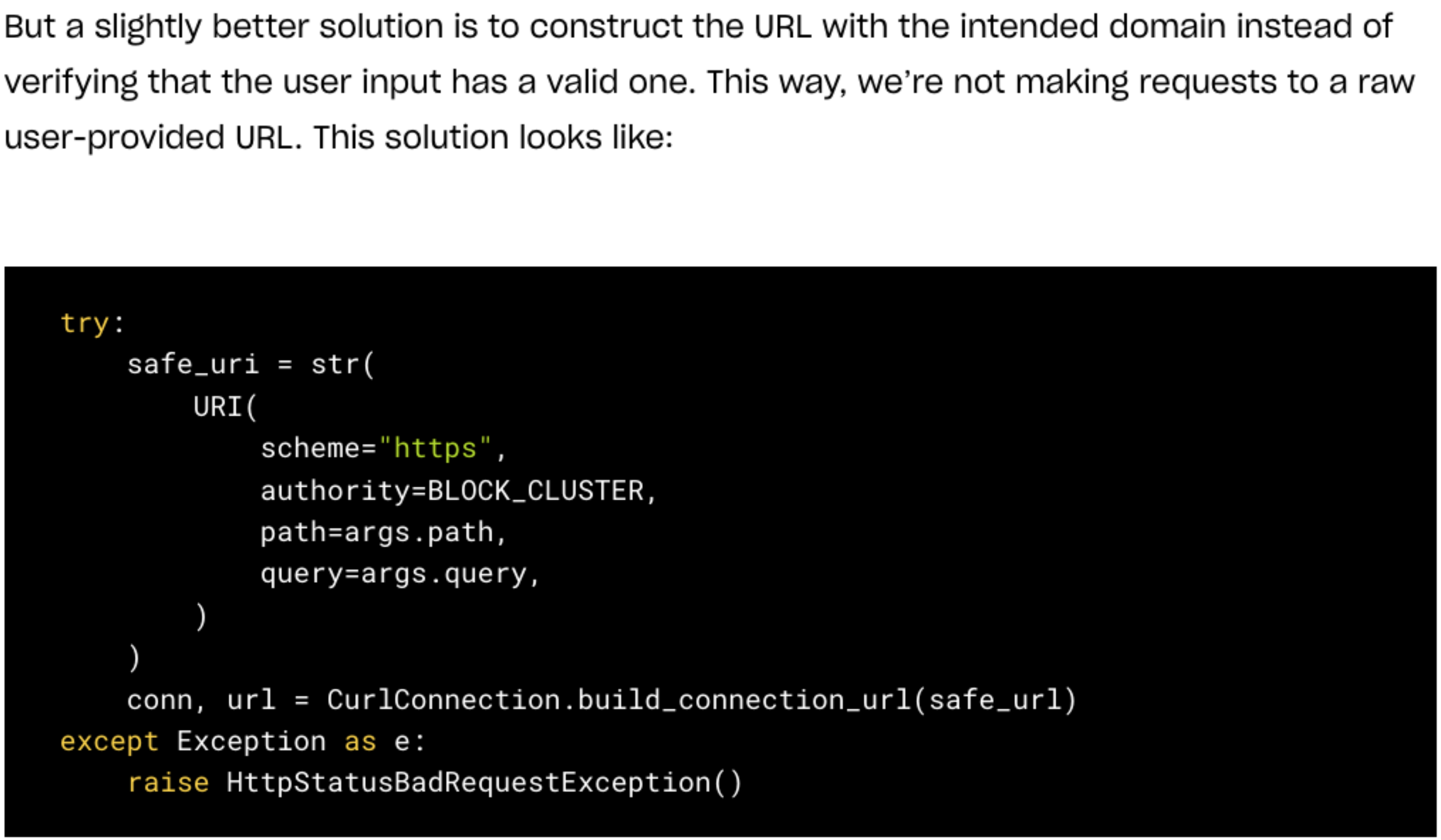
```
In [1]: URI.parse('https://dl-web.dropbox.com\@127.0.0.1:8080').authority
Out[1]: 'dl-web.dropbox.com'
```

However, parsing it with `urlsplit` would treat the part after `\@` as the hostname and direct the request to an attacker-specified address:

```
In [1]: urlsplit("https://dl-web.dropbox.com\@127.0.0.1:8080").hostname
Out[1]: '127.0.0.1'
```

https://dropbox.tech/security/bug-bounty-program-ssrf-attack

Fixing SSRF at Dropbox



But a slightly better solution is to construct the URL with the intended domain instead of verifying that the user input has a valid one. This way, we're not making requests to a raw user-provided URL. This solution looks like:

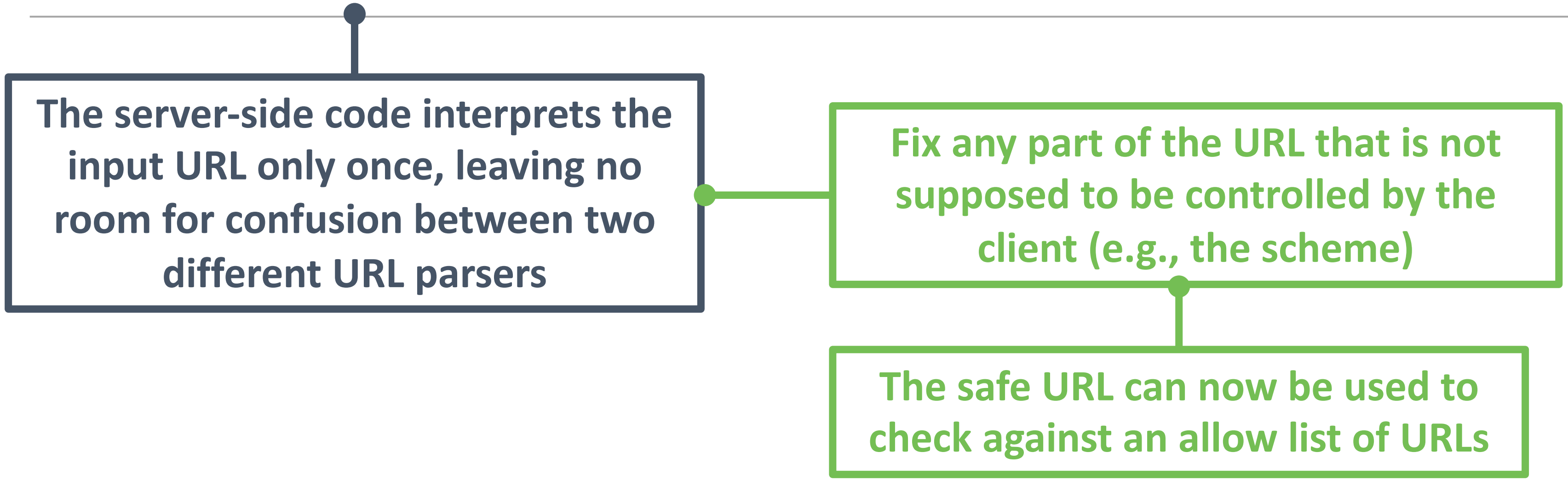
```
try:
    safe_uri = str(
        URI(
            scheme="https",
            authority=BLOCK_CLUSTER,
            path=args.path,
            query=args.query,
        )
    )
    conn, url = CurlConnection.build_connection_url(safe_url)
except Exception as e:
    raise HttpStatusBadRequestException()
```

https://dropbox.tech/security/bug-bounty-program-ssrf-attack

Fixing SSRF by removing ambiguity on the server

Accept a URL as input on the server and immediately transform it into a unambiguous value

```javascript
const input = req.body.url
// Transform the input into a single representation
const u = new URL(input)
// Construct a unambiguous safe URL to use in the application
const safeUrl = new URL(`https://${u.host}${u.pathname}${u.search}`)
// Check if ‘safeUrl’ is actually safe
```



The server-side code interprets the input URL only once, leaving no room for confusion between two different URL parsers

Fix any part of the URL that is not supposed to be controlled by the client (e.g., the scheme)

The safe URL can now be used to check against an allow list of URLs

Protect against SSRF

SSRF vulnerabilities often occur when there's ambiguity in matching against allow-lists.

Ensure the data used for server-side requests is unambiguous and trustworthy according to your security policy.

1. Broken object level authorization
2. Broken authentication
3. Broken object property-level authorization
4. Unrestricted resource consumption
5. Broken function level authorization
6. Unrestricted access to sensitive business flows
7. Server-side request forgery
8. Security misconfiguration
9. Improper inventory management
10. Unsafe consumption of APIs

OWASP

API Security

TOP10

MASS ACCOUNT TAKEOVER IN THE YUNMAI SMART SCALE API

https://fortbridge.co.uk/research/mass-account-takeover-yunmai/

> **Account takeover through ‘forgot password’ functionality.**
>
> **The victim will get an email with a unique 6 digit code that allows to reset the password.**

Breaking authentication

Avoid leaking information

APIs often (unknowingly) leak information that enables attacks such as username enumeration.

Carefully analyze your APIs for explicit and implicit data leakage.

Implement rate limiting

Many endpoints fail to implement rate limiting, which allows attackers to launch brute force attacks. Examples include SMS code prompts, reset tokens, and authentication forms.

Implement rate limiting to minimize the attacker's ability to abuse these endpoints.

Mitigate guessing attacks

Attackers often abuse unsigned values to implement guessing attacks.

Mitigation techniques against guessing attacks include using long random identifiers (e.g., a UUID) or using signed values that allow the detection of tampering.

1	Broken object level authorization
2	Broken authentication
3	Broken object property-level authorization
4	Unrestricted resource consumption
5	Broken function level authorization
6	Unrestricted access to sensitive business flows
7	Server-side request forgery
8	Security misconfiguration
9	Improper inventory management
10	Unsafe consumption of APIs

OWASP

API Security

TOP 10

MASS ACCOUNT TAKEOVER IN THE YUNMAI SMART SCALE API

https://fortbridge.co.uk/research/mass-account-takeover-yunmai/

> **The Android and iOS API were discovered to not implement any authorization checks while adding or deleting ‘family member’ accounts to/from other accounts.**

Why is authorization so hard to get right?

Enforcing role-based access control on controller endpoints in Spring

```java
@PreAuthorize("hasRole('FAMILY_OWNER')")
public void addMember(long familyId, FamilyMember member) {
  familyData.addMember(familyId, member);
};
```

RBAC typically leads to role explosion to express fine-grained permissions

```
@PreAuthorize("hasRole(FAMILY_OWNER')
            or hasRole('PARENT')
            or hasRole('ADMIN')")
public void addMember(long familyId, FamilyMember member) {
  familyData.addMember(familyId, member);
};
```

A policy like this is hard to maintain. Additionally, every change to the policy requires code changes to enforce this.

Permission-based security decouples the code from the authorization policy

```java
@PreAuthorize("hasPermission('ADD_FAMILY_MEMBER')")
public void addMember(long familyId, FamilyMember member) {
  familyData.addMember(familyId, member);
};
```

Permissions decouple user permissions from endpoints. Permissions are now mapped to a user (E.g., via roles, groups, …)

Hmm, that wasn't so hard?

Permission-based security decouples the code from the authorization policy

```
@PreAuthorize("hasPermission('ADD_FAMILY_MEMBER')")
public void addMember(long familyId, FamilyMember member) {
  familyData.addMember(familyId, member);
};
```

Adding a family member

```
POST /family/1/member HTTP/1.1
{ name: … }
```

This adds a new member to your family

Adding a family member

```
POST /family/7/member HTTP/1.1
{ name: … }
```

This adds a new member to someone else's family

A permission check only allows authorized users to access this endpoint

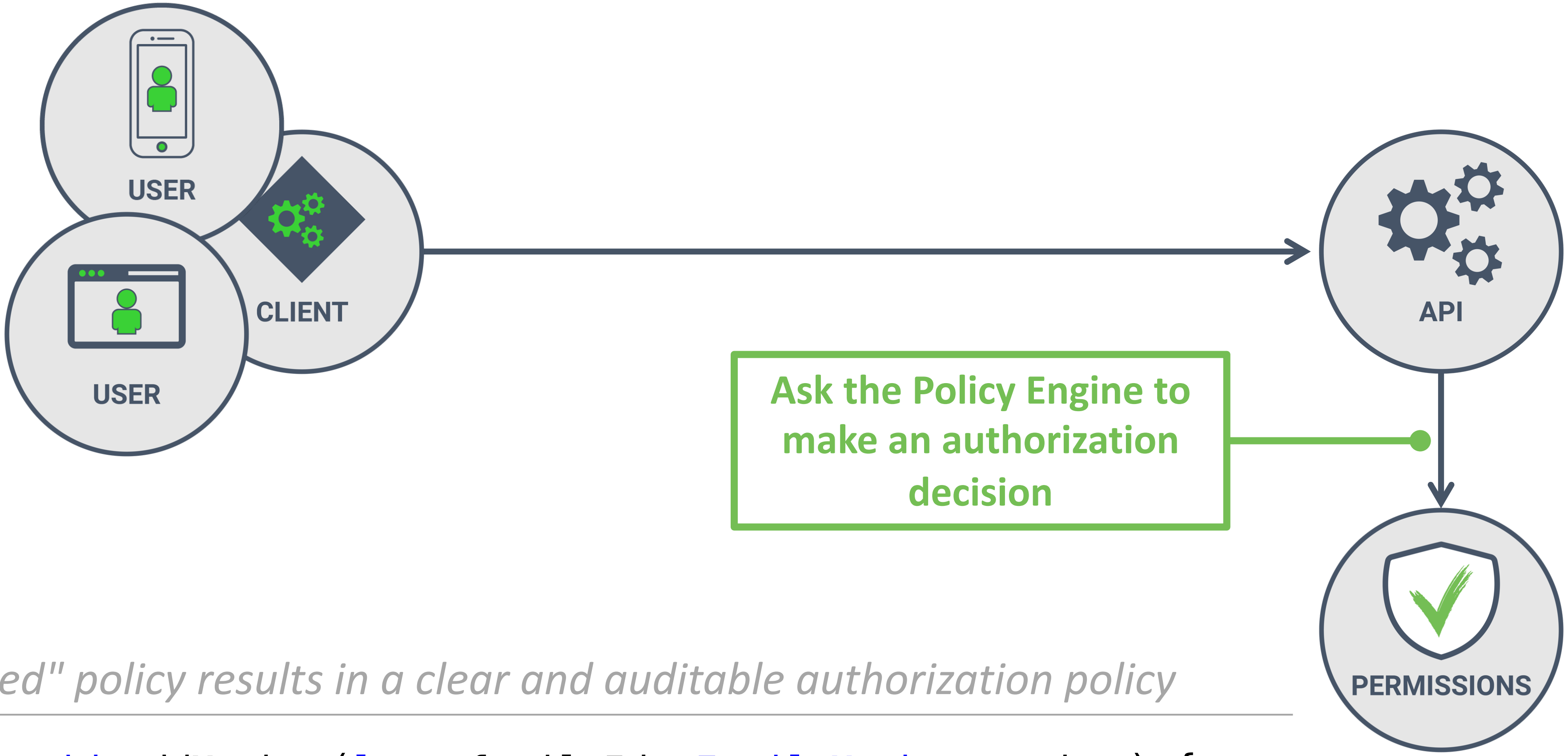
Object-level access control is often challenging to implement

```java
@PreAuthorize("hasPermission('ADD_FAMILY_MEMBER')")
public void addMember(long familyId, FamilyMember member) {
  Family f = familyData.get(familyId);
  if((user.hasRole("FAMILY_ONWER") || user.hasRole("PARENT")) && !f.isMember(user))) {
    throw new AuthorizationException(":(");
  }

  familyData.addMember(familyId, member);
};
```

Policies like these are impossible to audit for security

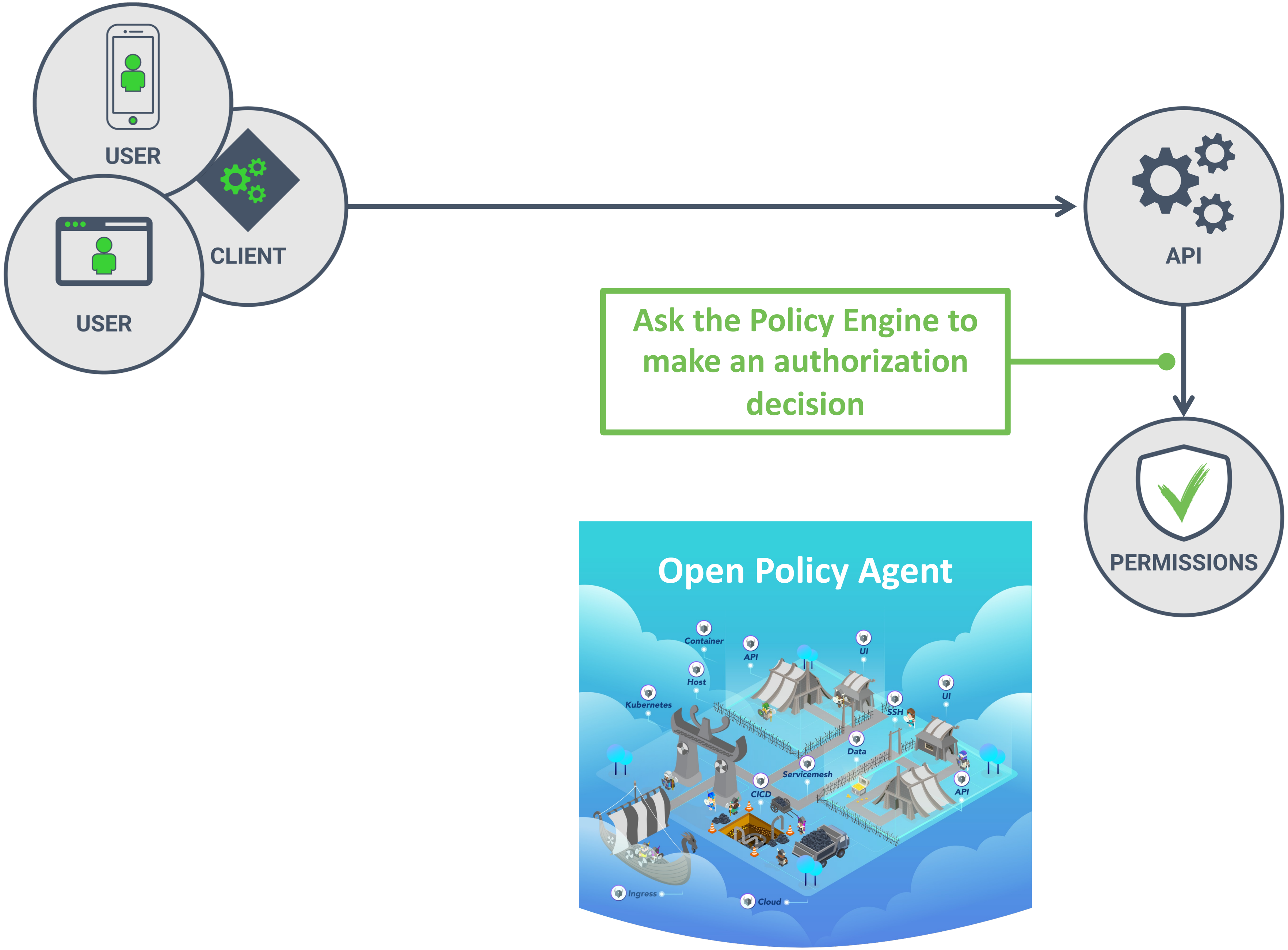
Certain roles require additional restrictions, such as being a member of the family



Ask the Policy Engine to make an authorization decision

A "centralized" policy results in a clear and auditable authorization policy

```java
public void addMember(long familyId, FamilyMember member) {
  Family f = familyData.get(familyId);
  if(policy.checkPermission("ADD_FAMILY_MEMBER"), f, user)) {
    familyData.addMember(familyId, member);
  }
  throw new AuthorizationException(":(");
}
```



Ask the Policy Engine to make an authorization decision

https://www.openpolicyagent.org/

Centralize complex authorization logic

Complex authorization logic should not be scattered throughout the code, but is best defined in a clear and understandable authorization policy

Empower auditability

Simplify the auditing of your authorization policy by making authorization logic explicit, even when endpoints have no specific authorization requirements.

1. Broken object level authorization
2. Broken authentication
3. Broken object property-level authorization
4. Unrestricted resource consumption
5. Broken function level authorization
6. Unrestricted access to sensitive business flows
7. Server-side request forgery
8. Security misconfiguration
9. Improper inventory management
10. Unsafe consumption of APIs

OWASP

API Security

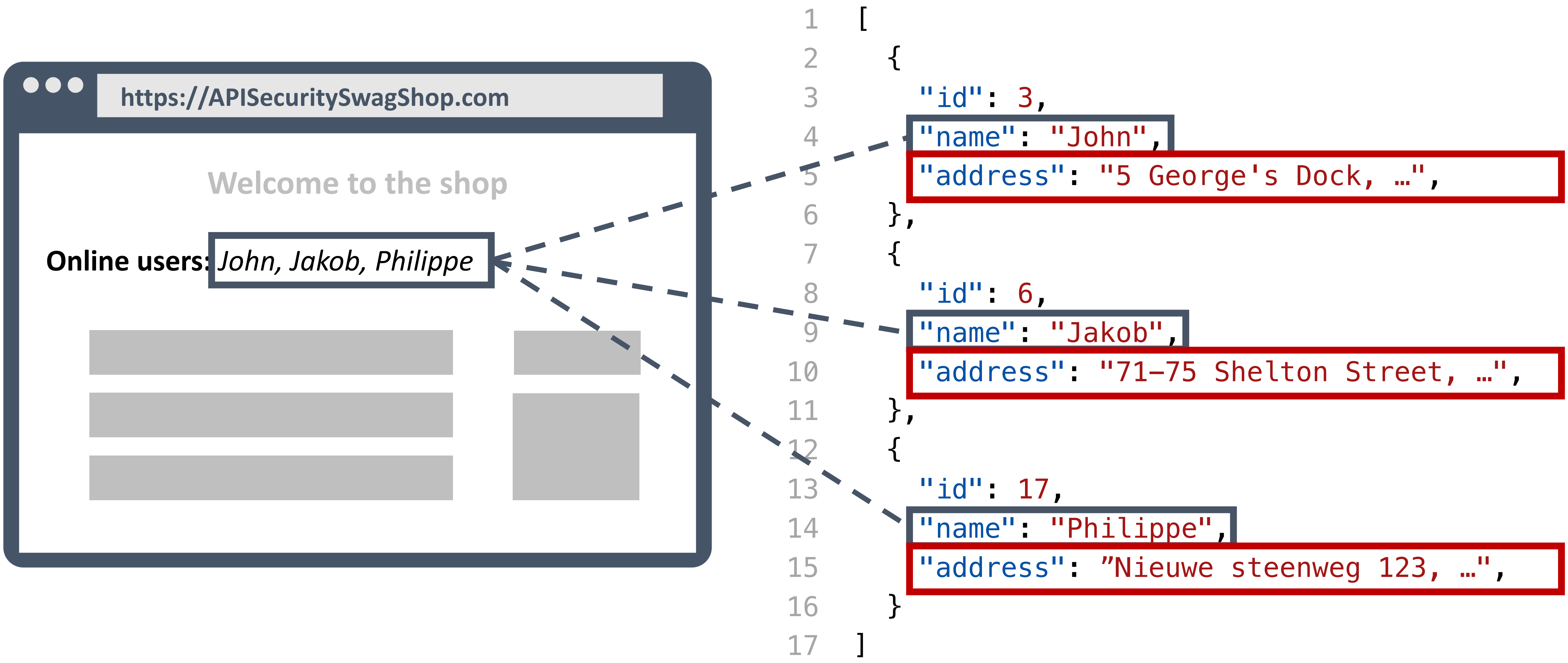
TOP10

MASS ACCOUNT TAKEOVER IN THE YUNMAI SMART SCALE API

https://fortbridge.co.uk/research/mass-account-takeover-yunmai/

> The server leaks the 'accessToken', and the 'refreshToken'. As a result, we can impersonate the account.

The API response to retrieve online users


https://APISecuritySwagShop.com
Welcome to the shop
Online users: *John, Jakob, Philippe*

```
[
  {
    "id": 3,
    "name": "John",
    "address": "5 George's Dock, …",
  },
  {
    "id": 6,
    "name": "Jakob",
    "address": "71–75 Shelton Street, …",
  },
  {
    "id": 17,
    "name": "Philippe",
    "address": ”Nieuwe steenweg 123, …",
  }
]
```

The Java Spring endpoint returning users

```java
@RequestMapping(path = "/online/users", method = GET, produces = "application/json")
public ResponseEntity<Object> getOnlineUsers() {
  List<User> users = UserService.getOnlineUsers();
  return new ResponseEntity<Object>(users, HttpStatus.OK);
}
```

The User data class

```java
public class User {
  private String id, name, address;
  …
  public String getName() {
    return name;
  }

  public String getAddress() {
    return address;
  }
}
```

Data fields are automatically translated to JSON, even when they are not supposed to be exposed

The Java Spring endpoint returning users

```java
@RequestMapping(path = "/online/users", method = GET, produces = "application/json")
public ResponseEntity<Object> getOnlineUsers() {
  List<User> users = UserService.getOnlineUsers();
  return new ResponseEntity<Object>(users, HttpStatus.OK);
}
```

The User data class

```java
public class User {
  private String id, name, address;

  public String getName() {
    return name;
  }

  @JsonIgnore
  public String getAddress() {
    return address;
  }
}
```

Annotations can hide fields, but this approach does not really follow the *deny-by-default* best practice

The Java Spring endpoint returning users

```
@RequestMapping(path = "/online/users", method = GET, produces = "application/json")
public ResponseEntity<Object> getOnlineUsers() {
  List<User> users = UserService.getOnlineUsers();
  return new ResponseEntity<Object>(users.stream().map(PublicUserInfo::new), HttpStatus.OK);
}
```

The PublicUserInfo DTO class

```
public class PublicUserInfo {
  private String id, name;

  public PublicUserInfo(User user) {
    this.setId(user.getId());
    this.setName(user.getName());
  }
  …
  public String getName() {
    return name;
  }
}
```

The DTO class only defines fields that are supposed to be exposed.

A User object is never directly exposed to the client.

Avoid sensitive data exposure

Avoid directly returning internal application data, as this often results in the exposure of sensitive data.

Use strict schemas or DTOs in combination with a well-defined OpenAPI specification of your API.

1 Broken object level authorization

2 Broken authentication

3 Broken object property-level authorization

4 Unrestricted resource consumption

5 Broken function level authorization

6 Unrestricted access to sensitive business flows

7 Server-side request forgery

8 Security misconfiguration

9 Improper inventory management

10 Unsafe consumption of APIs

OWASP

API Security

TOP10

Key takeaways

1. The best authorization policy is understandable and auditable

2. Analyze your APIs for data leakage and brute force attack vectors

3. Perimeter security cannot be your only defense

SecAppDev 2025

June 2 - 6, Leuven, Belgium

sis architecture
Rules
CST
AST/IL
Analysis
YAML patterns and metadata
tree-sitter parsers
Writing rules
Rules repository
Semgrep Playground
Semgrep Registry
Search mode:
Semantic-aware AST pattern matching
Single file analysis
Intraprocedural symbolic constant propagation
Understanding of types and external symbols
Autofix
Taint mode:
Extension of search mode
Intraprocedural taint propagation
Extends syntax with support for sources, sinks, sanitizers and propagators
Taint labels and composition
Extract mode:
Extract and analyse code embedded in a different language
Join mode:
Multi file analysis
Join results based on metavariables and file paths

More info and registration: https://secappdev.org

Thank you!

Reach out to discuss
how I can help you with security

https://pragmaticwebsecurity.com