A POSITIVE SECURITY MODEL FOR APIS

ISABELLE MAUNY
ISABELLE@42CRUNCH.COM
Introducing Security Models
NEGATIVE SECURITY MODEL (BLACKLIST)

Access **Allowed** by default

Block access for suspicious traffic

Threats **centric**
POSITIVE SECURITY MODEL (WHITELIST)

Access Denied by default

Allow Access only to approved traffic

Trust centric
WHY A POSITIVE MODEL?

- Much stricter access control
- Limited false positives
- More efficient
  - Simple vs. very complex regular expressions for blacklisting
- No need to update when new threats are found
However...
KEEPING UP IS HARD...

A whitelist is only powerful if complete!

It requires lots of efforts to define and maintain up to date with constant applications changes

✓ High human cost, usually several people full time

Traditionally been very hard to implement

✓ Which is why default WAF model is blacklisting
...BUT APIS ARE DIFFERENT!

- OpenAPI specification (OAS) can be leveraged to describe the **API contract**.
- Can be easily updated from code, or via specialized tools, so the whitelist is always in sync with the application.
- You can start addressing security straight from design time!
- OpenAPI lets you build the **ultimate whitelist**!
  - ✓ And as bonus, you get better documentation!
HOW 42CRUNCH LEVERAGES OAS

Audit Service performs 200+ security checks on API Contract

Scan service ensures API implementation conforms to API contract

Protection service is automatically configured from API contract
OWASP API SECURITY TOP 10

- **API1**: Broken Object Level Authorisation
- **API2**: Broken Authentication
- **API3**: Excessive Data Exposure
- **API4**: Lack of Resources & Rate Limiting
- **API5**: Missing Function/Resource Level Access Control
- **API6**: Mass Assignment
- **API7**: Security Misconfiguration
- **API8**: Injection
- **API9**: Improper Assets Management
- **API10**: Insufficient Logging & Monitoring
Addressing API threats with a positive model
The Attack

✓ Remote command injection attack: server executes commands written in ONGL language when a Content-Type validation error is raised.
✓ Can also be exploited using the Content-Disposition or Content-Length headers

POST / HTTP/1.1
Connection: Keep-Alive
Content-Type: %((#Normal='multipart/form-data').(#d=ongl.ongl0Context#DEFAULT_MEMBER_ACCESS).(#_memberAccess? (##_memberAccess##'##memberAccess?##(##context##'##context[##com.opensymphony.xwork2.ActionContext##.##container##'])).(#onglUtil=#container.getInstance(##com.opensymphony.xwork2.ongl.onglUtil##class)).(#onglUtil.getExcludedPackageNames().clear()).(#context.setMemberAccess(#d))).([#cmd='whoami']).
Accept: text/html, application/xhtml+xml, */*
Accept-Language: zh-CN

The Breach

✓ One of the most important in history: 147 millions people worldwide, very sensitive data
✓ Equifax got fined $700 million in Sept 2019

Core Issue

✓ Remote command injection vulnerability in Apache Struts widely exploited during months.
CONTENT-TYPE IN OAS

- Declare “consumes” at API or operation level

```json
"consumes": [
  "application/x-www-form-urlencoded",
  "application/json"
],
```

- Limits **Content-Type** header value to specific mime types

- Declare all request headers
HOW 42CRUNCH ADDRESSES THE PROBLEM

▷ At **Audit** time
  ✓ Detect that **Consumes** is not defined

▷ At **Scan** time
  ✓ Inject wrong Content-Type
  ✓ Inject wrong formats for all listed headers

▷ At **Runtime**
  ✓ Block any Content-Type that does not match Consumes value at Runtime
  ✓ Block any header not matching the description
  ✓ Block inbound data that does not match the Content-Type
HARBOUR REGISTRY


▶ The Attack
  ✓ Privilege escalation: become registry administrator

▶ The Breach
  ✓ 1300+ registries with default security settings

▶ Core Issue
  ✓ Mass Assignment vulnerability allows any normal user to become an admin

```
POST /api/users
{
  "username": "test",
  "email": "test123@gmail.com",
  "realname": "noname",
  "password": "Password1\0021",
  "comment": null,
  "has_admin_role": True
}
```
HOW OAS CAN BE USED?

- Describe inbound schema for all requests

- Use different schemas by operation (retrieve user data vs. update user data)

```json
"UsersItem": {
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "_id": {
      "type": "number",
      "format": "integer",
      "minimum": 0,
      "maximum": 999999
    },
    "email": {
      "type": "string",
      "format": "email",
      "pattern": "<email_regex>",
      "minLength": 10,
      "maxLength": 60
    },
    "is_admin": {
      "description": "is admin",
      "type": "boolean"
    },
    ...
  }
}
```
HOW 42CRUNCH ADDRESSES THE PROBLEM

▶ At Audit time
  ✓ Detects that schemas are not associated to requests
  ✓ Analyzes how well data is defined (patterns, min, max, enums)
  ✓ Highlights usage of “additional properties”

▶ At Scan time
  ✓ Injects additional properties
  ✓ Injects improper data

▶ At Runtime
  ✓ Enforces schema definition
  ✓ Enforces Additional Properties restrictions
  ✓ Block non-declared VERBs (block unwanted POST)
UBER (SEPT 2019)

- The Attack
  - Account takeover for any Uber account from a phone number

- The Breach
  - None. This was a bug bounty.

- Core Issues
  - First Data leakage: driver internal UUID exposed through error message!
    ```json
    {
      "status":"failure",
      "data": {
        "code":1009,
        "message":"Driver '47d063f8-0xx5e-xxxxx-b01a-xxx' not found"
      }
    }
    ```
  - Second Data leakage via the getConsentScreenDetails operation: full account information is returned, when only a few fields are used by the UI. This includes the **mobile token** used to login onto the account.
HOW OAS CAN BE USED?

- Describe thoroughly all potential responses
- Define the produces value
  - Which data will be returned
- Use different schemas by operation (retrieve user data vs. update user data)

```
"produces": [
  "application/json"
],

"responses": {
  "200": {
    "description": "successful..",
    "schema": {
      "type": "array",
      "minItems": 0,
      "maxItems": 50,
      "items": {
        "$ref": "#/definitions/UsersItem"
      }
    }
  },
  "403": {
    "description": "invalid...",
    "schema": {
      "type": "object",
      "properties": {
        "message": {
          "type": "string",
          "pattern": "xxxx",
          "minLength": 1,
          "maxLength": 255
        }
      },
      "success": ...
    }
  }
}
```
HOW 42CRUNCH ADDRESSES THE PROBLEM

At Audit time
- Analyzes which responses should be defined depending on verb (GET, POST, ...)
- Detects that schemas are not associated to responses
- Analyzes how well data is defined (patterns, min, max, enums)
- Highlights usage of “additional properties”

At Scan time
- Validates responses are all defined in contract
- Validates responses match schemas defined in contract

At Runtime
- Block responses that do not match “Produces” value (unknown mime-type)
- Blocks responses that do not match schema definition
- Block non-declared responses (unknown HTTP codes)
- Enforces Additional Properties restrictions
A POSITIVE MODEL FOR API SECURITY WITH 42CRUNCH

▶ Leverage OAS and build the ultimate whitelist at **design** time!
  ✓ Right in your IDE with our VSCode extension
  ✓ Thorough report with priorities to act upon

▶ Ensure API Contract is up to date via automated audit and scan at **integration/testing** time
  ✓ Include API Contract audit and scan in your favorite CI/CD pipeline

▶ Leverage the power of OAS to protect your APIs at runtime
  ✓ Lightweight, Kubernetes-ready firewall to automatically protect your APIs from API contract!
RESOURCES

- 42Crunch Website
- Free OAS Security Audit
- OpenAPI VS Code Extension
- OpenAPI Spec Encyclopedia
- OWASP API Security Top 10
- APIsecurity.io