**Zero Trust Auto-Remediation on AWS (REJECT Traffic Isolation)**

After completing my secure architecture and detection projects, I wanted to take things further. Not just logging or alerting, but **real cloud-native incident response**. This project simulates a real-world situation: when a protected EC2 instance receives **REJECT traffic**, the system will automatically detect it and **detach a lockdown security group**, effectively isolating the instance from the source of the threat.

**This is a project that mirrors what security teams want in production** — automation, speed, and tight controls.

**⚠️ *Note: This project was designed and tested in a lab environment. While it simulates real-world automation, production systems would require additional safeguards such as correlation logic, rate thresholds, rollback functions, and approval workflows.***

**💼 Real-World & Business Use Case**

If a server (especially one that holds sensitive data or backend services) is receiving network traffic that’s actively being blocked, something’s wrong.

Whether it’s a misconfiguration, a brute-force attack, or lateral movement — we need to respond. In a business environment, this response might come from a SIEM or analyst. In the cloud? We can **automate it**.

This system:

* Detects REJECT traffic in real-time
* Triggers CloudWatch
* Uses Lambda to remove dangerous SGs
* Leaves the instance in an isolated state
* **No human intervention required**

A diagram of a cloudwatch alarm

AI-generated content may be incorrect.

***Diagram showing the flow from VPC Flow Log REJECT entries → CloudWatch Alarm → Lambda response to auto-remove risky security group.***

**🧱 AWS Services Used**

|  |  |
| --- | --- |
| **Service** | **Purpose** |
| **VPC Flow Logs** | Capture REJECT traffic between EC2s |
| **CloudWatch Log Group** | Store logs for parsing |
| **Metric Filter** | Detect the word REJECT |
| **CloudWatch Alarm** | Trigger Lambda based on metric |
| **Lambda (Python 3.12)** | Parse log, find EC2, detach SG |
| **IAM Tags & Policies** | Restrict Lambda access and scope |

❌ **No SNS**  
❌ **No EventBridge**  
✅ **Direct CloudWatch → Lambda trigger**

**⚙️ What I Did (Action Flow)**

Below is a breakdown of what I built — not quite a tutorial, but everything I actually did:

**1. Created a Custom VPC**

* Made 3 subnets across different AZs
* One subnet was public (attached IGW + route table)
* Two subnets were private
* Associated route tables accordingly

A screenshot of a computer

AI-generated content may be incorrect.

**2. Created VPC Flow Logs**

* Chose destination: **CloudWatch Logs**
* Created log group: ZTFlowLogGroup
* Verified REJECT logs were appearing

A screenshot of a computer

AI-generated content may be incorrect.

**3. Deployed 2 EC2s**

* Public EC2 in public subnet (test attacker)
* Private EC2 (ServiceA-Instance) in private subnet
* Attached security group ZT-DenyAll to ServiceA-Instance to trigger REJECTs

📸 A screenshot of a computer

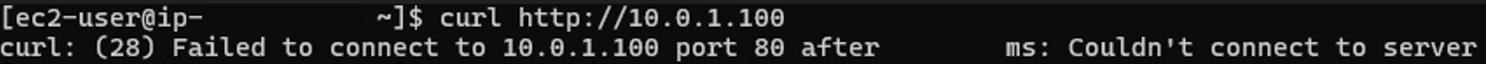
AI-generated content may be incorrect.

**4. Tested REJECT Traffic**

* SSH’d into public EC2
* Ran

1. curl http://10.0.1.100

* Confirmed REJECT traffic appeared in log group



**5. Created Metric Filter**

* Log group: ZTFlowLogGroup
* Filter pattern: REJECT
* Metric name: RejectedConnections
* Namespace: ZeroTrustMonitor

A screen shot of a computer

AI-generated content may be incorrect.

**6. Created CloudWatch Alarm (Before Lambda)**

Even before writing the Lambda, I set up the detection pipeline so I could simulate REJECT traffic and watch the alarm fire.

* Metric name: RejectedConnections
* Namespace: ZeroTrustMonitor
* Threshold: ≥ 1 datapoint
* Period: 10 seconds (so I could test quickly)
* **No action was assigned yet** — I hadn’t built the Lambda yet.

This let me test if REJECTs were being detected in real time using curl or SSH attempts from a public EC2.

A screenshot of a computer

AI-generated content may be incorrect.

**7. Created Lambda Function & Connected to Alarm**

Once the alarm worked, I moved on to remediation.

* Created Lambda (Python 3.12)
* IAM permissions added:

**DescribeNetworkInterfaces**

**DescribeInstances**

**ModifyNetworkInterfaceAttribute**

I **tested the Lambda manually** with a test payload, and it worked: it successfully removed the SG.

After that, I attached the Lambda to the alarm as its target action.

A screenshot of a computer

AI-generated content may be incorrect.

**8. Triggered the Full Pipeline (REJECT → Isolation)**

To confirm everything worked end-to-end, I SSH’d into my public EC2 and ran:

1. curl http://10.0.1.100

* REJECT traffic showed up in logs
* CloudWatch metric updated
* Alarm went into ALARM
* Lambda was triggered
* **ZT-DenyAll** was removed from the instance

A screenshot of a computer

AI-generated content may be incorrect.

**🔐 Why This Is Important**

This architecture reflects **real-world incident response**:

* Something suspicious happens
* You act immediately
* No humans required

It's also aligned with **Zero Trust** — we’re assuming that if traffic is rejected, it shouldn’t be allowed at all.

**✅ NIST / ISO Alignment**

|  |  |
| --- | --- |
| Control | Description |
| NIST AU-12 / ISO A.12.4.1 | Log monitoring and filtering |
| NIST IR-5 / ISO A.16.1.5 | Automated incident response |
| NIST AC-6 / ISO A.9.2.3 | Least privilege enforcement on network access |

**🧠 Errors & Fixes**

**❌ Metric Filter Not Triggering**

I forgot that it only evaluates logs AFTER it's created.  
**Fix**: Generated new traffic manually with curl. Eventually started firing.

**❌ Lambda Not Triggering**

I hadn’t attached Lambda directly to the alarm.

**Fix**: Went into CloudWatch Alarm → set action to **Lambda function**

**❌ SG Didn’t Get Removed**

Lambda had no permission. Also the EC2 wasn’t tagged.

**Fix**:

* Tagged EC2 as ServiceA-Instance
* Added IAM permissions: ModifyNetworkInterfaceAttribute, etc.

**🧹 Cleanup**

To stay within the Free Tier, I:

* Terminated all EC2s
* Deleted Lambda, flow logs, metric filter, and alarm
* Cleaned up IGW and route table

**🔁 Possible Real-World Upgrade**

If building this for a company:

* Replace hardcoded tag check with: AutoRemediate = true
* Send alert via Slack or SNS
* Store data in DynamoDB
* Add rollback logic
* Auto-create incident tickets

**📸 Screenshot List for Portfolio/GitHub**

1. Architecture diagram
2. Subnet + route tables
3. VPC Flow Log setup
4. Metric Filter config
5. CloudWatch Alarm config
6. EC2 instance tags
7. curl test result (REJECT)
8. Lambda logs showing REJECT and SG removal
9. SG removed from EC2

🧾 **Final Thoughts**

This project pushed me technically and mentally, and that’s exactly why I wanted to do it. Working with tightly connected AWS services like CloudWatch, Flow Logs, and Lambda taught me how even small misconfigurations can cause big problems — and more importantly, how to troubleshoot them under pressure.

I got the pipeline working as designed. It detects REJECT traffic, raises an alarm, and automatically isolates the instance by removing a restrictive security group. I also built in safety mechanisms, like instance tagging, to prevent unintended impact.

While I believe the project is functioning correctly, I’d still welcome a peer review. I’m always looking to improve, whether that’s simplifying logic, adding resilience, or aligning more closely with best practices.

What matters most to me is that this wasn’t just about ticking boxes. This was real-world thinking: build, test, break, fix, document. I treated it like a live SOC scenario, and it shows.

I’m proud of how far I’ve come and even more excited about where I can go next.