BYZANTINE FAULT-TOLERANT INTRUSION DETECTION SYSTEM

Live Demonstration Script

Presented by: **Amogh**

Project: Distributed Network Security with Byzantine Consensus

TABLE OF CONTENTS

1. PRE-DEMO SETUP

Terminal Configuration

I'm Amogh, and I'll be starting all systems in this sequence before the demo begins:

Terminal 1 - rp0 (Coordinator)

```
cd ~/byzantine_coordinator
python3 coordinator.py
```

Terminal 2 - rp6 (Primary Detector)

```
cd ~/byzantine_ids
python3 detector_bft.py
```

Terminal 3 - rp6 (Log Forwarder)

```
cd ~/byzantine_ids
python3 log_forwarder.py
```

Terminal 4 - rp8 (Physical Detector)

```
python3 ~/detector_bft.py
```

Terminal 5 - rp8 (Virtual Byzantine Node)

```
python3 ~/detector_virtual.py
```

■ Verify all terminals show 'Started' messages before proceeding.

2. INTRODUCTION & SYSTEM OVERVIEW

Opening Statement (30 seconds)

I'm Amogh. Good morning/afternoon, everyone. Today I'm demonstrating a Byzantine Fault-Tolerant Intrusion Detection System that I built using a Raspberry Pi cluster. The system can detect network attacks while tolerating one malicious or faulty detector node. This is based on the Byzantine Generals Problem - a fundamental challenge in distributed computing where some nodes may provide false information.

System Architecture Explanation

I'm Amogh, and I've deployed 4 Raspberry Pi nodes in this setup:

Node	Role	Function
rp0	Byzantine Coordinator	Aggregates votes and reaches consensus
rp6	Primary Suricata IDS	Detects threats using 46,825+ signatures
rp8 Physical	Honest Voting Node	Independently verifies all detections
rp8 Virtual	Byzantine Faulty Node	Lies 30% of the time to test fault tolerance

3. TEST 1 - PORT SCAN DETECTION

Attack Command (Kali Terminal)

```
sudo nmap -sS 192.168.1.237
```

What I (Amogh) Will Explain:

I'm launching a TCP SYN stealth scan - this is a common reconnaissance technique attackers use to discover open ports without completing the TCP handshake. Watch all three detector terminals.

Point to Detectors:

All three detectors immediately see the attack:
*• **rp6** detected it through Suricata's signature matching*
*• **rp8 physical** confirmed the detection independently*
*• **rp8-virtual** also detected it (may vote honestly or lie)*

Point to Coordinator:

The coordinator shows consensus results:
*• If **3/2 votes** → all nodes voted honestly*
*• If **2/2 votes** → Byzantine node lied, but 2 honest nodes achieved consensus anyway*

Either result demonstrates the system working correctly.

4. TEST 2 - AGGRESSIVE SERVICE SCAN

Attack Command (Kali Terminal)

```
sudo nmap -T4 -A -v 192.168.1.237
```

What I (Amogh) Will Explain:

I'm running an aggressive scan with service version detection and OS fingerprinting. This is more intrusive and generates multiple types of alerts simultaneously.

Point to Detectors:

Notice multiple alert types appearing:
• Port Scan Detected
• Multiple Connection Attempts
• Possible SYN Flood

The system processes each threat type independently.

Point to Coordinator:

The coordinator processes each alert type separately and reaches consensus for each one. This demonstrates the system handling multiple simultaneous threats. Watch for yellow 'FAKE_CUSTOM ATTACK' lines - these indicate Byzantine lying behavior.

5. TEST 3 - PORT RANGE SWEEP

Attack Command (Kali Terminal)

```
sudo nmap -p 1-1000 192.168.1.237
```

What I (Amogh) Will Explain:

This is a port sweep scanning the first 1000 ports - another common reconnaissance technique. I'll now demonstrate the Byzantine node's behavior pattern clearly.

Point to rp8-virtual Terminal:

The virtual detector has 30% probability of lying on each alert:
*• **Blue panel**: Honest vote (70% probability)*
*• **Red panel with 'Lying!'**: Byzantine behavior (30% probability)*

When the red panel appears, you'll see:
• Real alert: Port Scan Detected
• Fake report sent: FAKE_CUSTOM ATTACK

Point to Coordinator:

But look at the coordinator - consensus was still reached with the 2 honest nodes. This is Byzantine Fault Tolerance in action. The faulty node's false information is automatically overridden by the honest majority.

6. TEST 4 - RAPID CONNECTION ATTEMPTS

Attack Command (Kali Terminal)

```
for i in {1..50}; do nc -zv 192.168.1.237 22 2>&1 | grep -i succeeded;
done
```

What I (Amogh) Will Explain:

This simulates a brute-force attack attempt by making 50 rapid SSH connection attempts. This is how attackers try to guess passwords through repeated login attempts.

Point to All Screens:

The deduplication algorithm prevents alert flooding - even though I'm making 50 connections, the system intelligently groups them and reports 'Multiple Connection Attempts' as a single threat classification. This prevents the coordinator from being overwhelmed.

Point to Coordinator:

Notice how quickly consensus is reached - typically under 1 second from detection to decision. This real-time processing is critical for automated threat response systems.

7. TEST 5 - SYN FLOOD (DOS ATTACK)

Attack Command (Kali Terminal)

```
sudo hping3 -S --flood -p 22 192.168.1.237
```

■■ Let it run for 3-5 seconds then press Ctrl+C

What I (Amogh) Will Explain:

This is a SYN flood attack - a denial of service technique that overwhelms the target with connection requests. This can crash vulnerable servers or make services unavailable to legitimate users.

Point to Detectors:

All detectors immediately flag this as 'Possible SYN Flood' - a critical severity alert. Even under this flood of malicious traffic, all three nodes maintain their detection capability and voting function.

Point to Coordinator:

Even under flood conditions, the Byzantine consensus system continues to function correctly. The distributed architecture prevents a single point of failure - if one detector was overwhelmed, the other two would maintain security coverage.

After Stopping:

I'm stopping the flood to prevent actual service disruption during this demonstration. In a production environment, this alert would trigger automated blocking rules.

8. BYZANTINE FAULT TOLERANCE PROOF

Demonstration Command (Kali Terminal)

```
for i in {1..5}; do sudo nmap -sS 192.168.1.237; sleep 2; done
```

What I (Amogh) Will Explain:

I'm running the same port scan 5 times in a row. With the Byzantine node's 30% lying probability, this will clearly demonstrate the fault tolerance pattern.

Point to rp8-virtual Terminal:

Watch the virtual detector's behavior across multiple attacks:
• ***Blue honest alerts*** *- node voting truthfully*
• ***Red Byzantine lies*** *- node sending false information*

This randomness simulates a real compromised node that behaves unpredictably.

Point to Coordinator:

Notice the coordinator's consistent response:
• *When Byzantine node is honest →* ***3/2 consensus*** *(all agree)*
• *When Byzantine node lies →* ***2/2 consensus*** *(2 honest nodes override the lie)*

Either way, the system produces the correct threat assessment. *This is the core principle of Byzantine Fault Tolerance - the system continues operating correctly even when some nodes are faulty or malicious.*

9. TECHNICAL DEEP DIVE

Core Principles (What I, Amogh, Will Explain):

The system implements Byzantine Fault Tolerance using these key principles:

Principle	Implementation	Benefit
1. Redundancy	3 independent detectors analyzing same traffic	No single point of failure
2. Voting Mechanism	Each detector casts vote to coordinator	Democratic decision making
3. Consensus Threshold	2 out of 3 votes required (majority)	Overrides faulty/malicious nodes
4. Fault Tolerance Formula	f = (n-1)/3, with n=3 → tolerates 1 fault	Mathematically proven resilience
5. Real-time Processing	Consensus reached within 1 second	Immediate threat response

Technology Stack:

- **Suricata IDS 7.0.1** - Industry-standard intrusion detection engine
- **46,825+ Signatures** - Emerging Threats ruleset for comprehensive coverage
- **Custom Detection Rules** - Tailored for laboratory environment
- **Python Flask API** - Lightweight vote collection service
- **UDP Log Forwarding** - Real-time alert distribution mechanism
- **Byzantine Consensus Algorithm** - Majority voting with fault tolerance
- **Rich Console Library** - Professional visualization and monitoring

10. ANTICIPATED QUESTIONS & ANSWERS

Q1: What happens if 2 nodes lie?

I'm Amogh, and with my current 3-node setup, 2 lying nodes would prevent consensus from being reached. To tolerate 2 Byzantine faults, I would need at least 7 nodes using the formula n = 3f+1, where f is the number of faults tolerated. This is a known limitation of Byzantine systems.

Q2: How did you simulate the Byzantine node?

The rp8-virtual detector uses Python's random number generator with 30% probability. When triggered, it prepends 'FAKE_' to the alert message before sending the vote. This simulates a compromised node that occasionally sends false information.

Q3: What's the performance impact?

In my testing, consensus is reached within 1 second. The main overhead is network communication - each alert requires 3 votes to be transmitted to the coordinator. However, this latency is minimal compared to the benefit of fault tolerance.

Q4: Could this work in production?

Yes, with several modifications: use production WSGI server instead of Flask development server, implement authentication between nodes, add encrypted communication channels, scale to more nodes for higher fault tolerance, and integrate with SIEM systems.

Q5: How does this compare to traditional IDS?

Traditional IDS has a single point of failure - if the IDS is compromised, attackers can blind the entire system. My Byzantine approach provides defense-in-depth through redundancy and voting, making it significantly harder for attackers to disable detection.

Q6: What about false positives?

Byzantine consensus actually helps reduce false positives in some cases. If one detector malfunctions and generates false alerts, the other two honest nodes will not confirm it, preventing false consensus. However, if all nodes agree on a false positive, the system will still report it.

CONCLUSION & KEY TAKEAWAYS

Closing Statement (What I, Amogh, Will Say):

This project successfully demonstrates:

■	Distributed Intrusion Detection	Multiple independent nodes working collaboratively
■	Byzantine Fault Tolerance	System functions correctly even with malicious nodes
■	Multiple Attack Detection	Port scans, connection floods, DoS attempts all detected
■	Real-world Applicability	Architecture scales to enterprise and cloud deployments

Final Statement:

The key innovation of my project is combining traditional signature-based intrusion detection with Byzantine consensus algorithms to create a more resilient security system. By distributing trust across multiple independent nodes and using majority voting, the system can tolerate compromise or failure of individual components while maintaining accurate threat detection. Thank you for your attention.

- Amogh