SEAS-8414 Week 09 Student Lab Guide

SEAS-8414 Week 09 Student Lab Guide: Supply Chain Integrity and Counterfeit Detection

Start Here: Download the Student ZIP

Before running any lab commands, download the Week 09 student package from the course site:

https://8414.bwater.io/downloads/labs/packages/seas8414-blackboard-week-09-2026.05.0_4e76a52f_aws8414.zip

The recommended path is the package Makefile:

```
unzip seas8414-blackboard-week-09-2026.05.0_4e76a52f_aws8414.zip
cd seas8414-blackboard-week-09-2026.05.0_4e76a52f_aws8414
make week09
```

The Makefile calls `run-week09-lab.sh`, extracts the nested runtime ZIP, starts the lab, runs the Week 09 API workflow, saves evidence under `lab-results/week-09/evidence/`, generates `lab-results/week-09/index.html`, and cleans up containers at exit.

You can also download the runner directly from https://8414.bwater.io/downloads/labs/scripts/run-week09-lab.sh.

Manual extraction uses two ZIP layers. First extract the weekly Blackboard ZIP, then extract the nested runtime ZIP:

```
unzip seas8414-blackboard-week-09-2026.05.0_4e76a52f_aws8414.zip
cd seas8414-blackboard-week-09-2026.05.0_4e76a52f_aws8414
unzip runtime/seas8414-student-lab-2026.05.0+4e76a52f.aws8414.zip
cd seas8414-student-lab-2026.05.0+4e76a52f.aws8414/student-lab
```

Run the instructions in this guide from that `student-lab/` directory. The matching screencast and LLM prompt are published next to the ZIP on the labs page:

- Screencast MP4: https://8414.bwater.io/downloads/labs/screencasts/phase09-lab-screencast.mp4
- LLM Prompt: https://8414.bwater.io/downloads/labs/prompts/phase09-lab-llm-prompt.md
- Run Script: https://8414.bwater.io/downloads/labs/scripts/run-week09-lab.sh

---

Breakwater Phase 9: Supply Chain Integrity & Counterfeit Detection Lab

Phase 9 introduces supply chain integrity analysis: software bill of materials (SBOM) extraction with transitive vulnerability tracing, multi-signal counterfeit detection (MAC OUI mismatch, firmware hash deviation, TCP stack fingerprinting, weak RNG entropy), vendor trust scoring across 10 dimensions, EU Cyber Resilience Act (CRA) compliance assessment, binary birthmark analysis, firmware phylogenetic tree construction, and Monte Carlo supply-chain attack simulation. These capabilities shift Breakwater from detecting operational vulnerabilities to evaluating whether the devices themselves can be trusted.

(See Slides 001-005 for Phase 9 overview, supply chain threat landscape, and trust model)

Prerequisites

- Completed Phase 1 lab (scan data with identified devices)
- Completed Phase 4 lab (attack graph and BRS scores computed)
- A completed scan with `$SCAN_ID` and `$TOKEN` variables set (see Phase 1 Exercise 6)
- Basic understanding of software package management (npm, pip, apt)
- Familiarity with EU cybersecurity regulation concepts

If you need to set up your variables from a previous session:

```
# Login and capture token
TOKEN=$(curl -s -X POST http://localhost:8100/v1/auth/login \
  -H "Content-Type: application/json" \
  -d '{"email":"student@example.com","password":"SecurePass!2026"}' \
  | jq -r '.access_token')

# Get the most recent completed scan ID
SCAN_ID=$(curl -s "http://localhost:8100/v1/scanning/smart-
        scan/history?limit=1" \
  -H "Authorization: Bearer $TOKEN" \
  | jq -r '.scans[0].scan_id')

echo "Token: ${TOKEN:0:20}..."
echo "Scan ID: $SCAN_ID"
```

What you should see:

```
Token: eyJhbGciOiJIUzI1Ni...
Scan ID: a3f1c2d4-5e6f-7890-abcd-ef1234567890
```

> **Troubleshooting:** If `Token: null` appears, verify the user account exists. If `Scan ID: null`, no completed scan exists -- run `python scan_report.py 172.30.0.0/24` to create one.

Phase 9 API Cheatsheet

Endpoint	Method	Description
`/v1/supply-chain/{scan_id}`	GET	SBOM analysis summary, vendor trust summary, and coverage metrics
`/v1/supply-chain/{scan_id}/counterfeit`	GET	Counterfeit detection findings
`/v1/supply-chain/{scan_id}/compliance`	GET	EU CRA compliance results
`/v1/supply-chain/{scan_id}/firmware`	GET	Firmware integrity check results
`/v1/supply-chain/{scan_id}/simulate`	POST	Monte Carlo supply-chain attack simulation

All Phase 9 endpoints require Bearer token authentication.

(See Slides 081-090 for API design, endpoint reference, and database models)

Exercise 1: Analyse the SBOM

(See Slides 010-022 for SBOM concepts, SPDX vs. CycloneDX formats, dependency graph construction, and diamond dependency detection)

A Software Bill of Materials (SBOM) is a machine-readable inventory of every software component in a device's firmware. It includes direct dependencies (libraries explicitly referenced by the firmware) and transitive dependencies (libraries that those libraries depend on). Transitive vulnerabilities -- CVEs in indirect dependencies -- are the most commonly missed class of vulnerability in IoT devices.

Architecture: What This Exercise Tests

```
graph LR
    subgraph "SBOM Analyser (sbom_analyzer.py)"
        A[Firmware Banner<br/>from Enrichment] --> B[Component
```

```
Extractor]
        C[HTTP Headers<br/>X-Powered-By etc.] --> B
        D[Service Fingerprints<br/>from Phase 2] --> B
        B --> E[DependencyNode tree]
    end
    subgraph "Dependency Graph (dep_graph.py)"
        E --> F[Diamond dep detection]
        E --> G[Transitive CVE lookup]
        F & G --> H[SBOMAnalysis]
    end
    subgraph "API"
        H --> I["GET /v1/supply-chain/{scan_id}"]
    end
```

Step 1: Get the SBOM summary for the scan

```
# Full SBOM summary
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" | jq .

# Key metrics
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '{hosts_analysed: .data.hosts_analysed,
         devices_with_sbom: .data.devices_with_sbom,
         total_components: .data.total_components,
         total_sbom_vulnerabilities:
        .data.total_sbom_vulnerabilities}'
```

What you should see:

```
{
  "hosts_analysed": 20,
  "devices_with_sbom": 12,
  "total_components": 187,
  "total_sbom_vulnerabilities": 43
}
```

Step 2: Inspect per-device SBOM data

```
# List devices with their component counts and vulnerability counts
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.analyses[] | select(.sbom_data.component_count > 0) |
        {ip: .device_ip, format: .sbom_format,
         components: .sbom_data.component_count,
         max_depth: .sbom_data.max_depth,
         transitive_cves: .sbom_data.transitive_cves_count,
         sbom_vulns: .sbom_vulnerabilities}]
        | sort_by(-.sbom_vulnerabilities)'
```

```
# Find devices with no SBOM (blind spots)
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.analyses[] | select(.sbom_data.component_count == 0) |
        .device_ip]'
```

What you should see for devices with SBOM:

```
[
  { "ip": "172.30.0.30", "format": "cyclonedx", "components": 48,
        "max_depth": 4,
    "transitive_cves": 12, "sbom_vulns": 15 },
  { "ip": "172.30.0.31", "format": "cyclonedx", "components": 45,
        "max_depth": 3,
    "transitive_cves": 10, "sbom_vulns": 13 },
  { "ip": "172.30.0.10", "format": "spdx",      "components": 22,
        "max_depth": 3,
    "transitive_cves": 5,  "sbom_vulns": 7  }
]
```

What you should see for devices without SBOM:

```
["172.30.0.40", "172.30.0.41", "172.30.0.42"]
```

PLCs (172.30.0.40-42) typically do not expose firmware component information via network banners, making SBOM extraction impossible without physical access.

Step 3: Detect diamond dependencies

```
# Find devices with diamond dependency conflicts
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.analyses[] | select((.sbom_data.diamond_deps // []) |
        length > 0) |
       {ip: .device_ip, diamond_count: (.sbom_data.diamond_deps |
       length),
        examples: .sbom_data.diamond_deps[:2]}]'
```

What you should see:

```
[
  {
    "ip": "172.30.0.30",
    "diamond_count": 3,
    "examples": [
      ["openssl", "1.1.1w", "1.1.1t"],
      ["libcurl", "7.88.1", "7.87.0"]
    ]
  }
]
```

A diamond dependency occurs when two different components in the dependency tree require different versions of the same library (e.g., openssl 1.1.1w and 1.1.1t). This can cause unpredictable behaviour and means two different CVE surfaces may apply to the same device.

> **Troubleshooting:** If `devices_with_sbom: 0`, the SBOM extractor did not find any component information in banners or headers. In the IoT simulation, SBOM data is inferred from device type and firmware version strings. Check that firmware version banners were captured: `curl -s http://localhost:8100/v1/analytics/behavioral/$SCAN_ID -H "Authorization: Bearer $TOKEN" | jq '[.data.baselines | to_entries[] | select(.value.firmware_version != null) | {ip: .key, firmware: .value.firmware_version}]'`.

Questions:

1. Of 20 devices analysed, 12 have SBOM data and 8 do not. What types of devices lack SBOM data (hint: look at which IPs are in the "no SBOM" list)? What are the security implications of a device with no SBOM for supply chain risk assessment? *(See Slides 013-015 for SBOM coverage challenges in embedded systems)*
2. The NAS at 172.30.0.30 has 48 components at depth 4. A depth of 4 means some components are 4 hops away from the direct dependency. Explain why transitive vulnerabilities at depth 3-4 are frequently exploited in practice: what makes them hard to detect without SBOM tooling? *(See Slide 017 for transitive dependency attack chains)*
3. A diamond dependency conflict `["openssl", "1.1.1w", "1.1.1t"]` means the device resolves to one version at runtime. The CVE surface of 1.1.1t includes vulnerabilities fixed in 1.1.1w. How do you determine which version is actually loaded, and what does this mean for your CVE count? *(See Slide 019 for diamond dependency resolution and security implications)*
4. The SBOM format for the NAS is `cyclonedx` and for the camera is `spdx`. Both are OASIS standards for SBOM interchange. What are the key differences between these formats? Which is better suited for regulatory compliance (EU CRA, US EO 14028)? *(See Slides 020-022 for SPDX vs. CycloneDX comparison)*

Exercise 2: Detect Counterfeit Devices

(See Slides 023-035 for counterfeit detection methodology, five signal types, and verdict computation)

Counterfeit IoT devices are physically identical to genuine products but manufactured without authorisation, often with inferior components, backdoored firmware, or deliberate security weaknesses. Breakwater's counterfeit detector uses five signals to classify devices: MAC OUI mismatch (MAC address assigned to wrong manufacturer), firmware hash mismatch (firmware differs from vendor's official hash), TCP stack deviation (OS fingerprint deviates from known-good profile), weak RNG entropy (random number generator produces low-entropy outputs), and timing anomaly (response timing inconsistent with known hardware).

Architecture: What This Exercise Tests

```
graph LR
    subgraph "Counterfeit Detector (counterfeit_detector.py)"
        A[MAC Address] --> B1[OUI DB lookup]
        C[Firmware Version] --> B2[Hash DB lookup]
        D[TCP SYN/ACK<br/>fingerprint] --> B3[Stack comparator]
        E[TLS Hello RNG] --> B4[Entropy analyser]
        F[RTT measurements] --> B5[Timing model]
    end
    subgraph "Verdict Engine"
        B1 --> S1[mac_oui_mismatch<br/>weight 0.3]
        B2 --> S2[firmware_hash_mismatch<br/>weight 0.35]
        B3 --> S3[stack_deviation<br/>weight 0.2]
        B4 --> S4[weak_rng<br/>weight 0.1]
        B5 --> S5[timing_anomaly<br/>weight 0.05]
        S1 & S2 & S3 & S4 & S5 --> V[CounterfeitVerdict]
        V -->|score < 30| W[genuine]
        V -->|score 30–60| X[suspicious]
        V -->|score > 60| Y[counterfeit]
    end
```

Step 1: Get all counterfeit detection results

```
# Full counterfeit detection results
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/counterfeit \
  -H "Authorization: Bearer $TOKEN" | jq .

# Summary: genuine vs. suspicious vs. counterfeit
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/counterfeit \
  -H "Authorization: Bearer $TOKEN" \
  | jq '{total_flagged: .data.total_flagged,
         counterfeit: .data.counterfeit,
         suspicious: .data.suspicious,
         results: [.data.results[] |
           {ip: .device_ip, verdict:
          .analysis_data.counterfeit.verdict,
             score: .analysis_data.counterfeit.trust_score}]
           | sort_by(-.score)}'
```

What you should see:

```
{
  "total_flagged": 4,
  "counterfeit": 1,
  "suspicious": 3,
  "results": [
    { "ip": "172.30.0.14", "verdict": "counterfeit",  "score": 72.0 },
    { "ip": "172.30.0.13", "verdict": "suspicious",   "score": 45.0 },
```

```
    { "ip": "172.30.0.20", "verdict": "suspicious",   "score": 38.0 },
    { "ip": "172.30.0.21", "verdict": "suspicious",   "score": 31.0 }
  ]
}
```

Step 2: Inspect counterfeit indicators for the flagged device

```
# Detailed indicators for the counterfeit-classified device
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/counterfeit \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[] | select(.analysis_data.counterfeit.verdict
        == "counterfeit") |
        {ip: .device_ip, score:
        .analysis_data.counterfeit.trust_score,
         indicators: [.analysis_data.counterfeit.indicators[] |
           {type: .indicator_type, confidence: .confidence, details:
        .details, severity: .severity}]}]'
```

What you should see:

```
[
  {
    "ip": "172.30.0.14",
    "score": 72.0,
    "indicators": [
      { "type": "mac_oui_mismatch",       "confidence": 0.9,
        "severity": "high",
        "details": "OUI 00:1A:2B registered to Hikvision but device
        identifies as Dahua" },
      { "type": "firmware_hash_mismatch", "confidence": 0.85,
        "severity": "high",
        "details": "Firmware hash a3f1c2d4 not in Hikvision official
        hash DB" },
      { "type": "stack_deviation",        "confidence": 0.7,
        "severity": "medium",
        "details": "TCP window size 8192 deviates from Hikvision V5
        baseline (65535)" }
    ]
  }
]
```

Step 3: Compare indicator profiles across device categories

```
# What indicators appear most frequently across all flagged devices?
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/counterfeit \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[].analysis_data.counterfeit.indicators[]
        | .indicator_type] | group_by(.) | map({type: .[0], count:
        length})'

# Compare suspicious vs. genuine devices' indicator counts
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/counterfeit \
```

```
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[] |
        {ip: .device_ip,
         verdict: .analysis_data.counterfeit.verdict,
         indicator_count: (.analysis_data.counterfeit.indicators |
        length),
         score: .analysis_data.counterfeit.trust_score}]'
```

What you should see for indicator frequency:

```
[
  { "type": "firmware_hash_mismatch", "count": 4 },
  { "type": "mac_oui_mismatch",       "count": 3 },
  { "type": "stack_deviation",        "count": 2 },
  { "type": "timing_anomaly",         "count": 1 },
  { "type": "weak_rng",               "count": 0 }
]
```

> **Troubleshooting:** If `total_flagged: 0`, the counterfeit detector found no anomalies. In the IoT simulation, counterfeit detection uses a simulated hash database. Device 172.30.0.14 is pre-configured as a counterfeit camera in the simulation. If your scan did not include this device, run `docker compose ps` and `./wait-for-lab.sh` to ensure all simulated devices are running.

Questions:

1. Device 172.30.0.14 has three counterfeit indicators: MAC OUI mismatch, firmware hash mismatch, and TCP stack deviation. Explain the significance of each indicator individually. Which one alone would be sufficient evidence to declare a device counterfeit, and why? *(See Slides 026-030 for individual indicator analysis and weight rationale)*
2. The MAC OUI for 172.30.0.14 is registered to Hikvision but the device identifies as Dahua. How do counterfeit manufacturers obtain legitimate OUIs? What is MAC spoofing, and why does OUI mismatch alone (without corroborating signals) have limited evidentiary value? *(See Slide 027 for MAC OUI mismatch analysis)*
3. The firmware hash mismatch indicator has `confidence: 0.85`. The hash is checked against the vendor's official firmware hash database. Name three scenarios that could produce a legitimate firmware hash mismatch that does NOT indicate counterfeit: (a) modified firmware, (b) partial update, (c) custom build. *(See Slide 028 for firmware hash mismatch false positive analysis)*
4. Weak RNG entropy (the `weak_rng` indicator) was not triggered for any device in this scan, but it appears in the indicator schema. Describe what weak RNG means for IoT security: what specific attack does low-entropy TLS random number generation enable? Name a real-world IoT device that was found to have weak RNG. *(See Slides 031-032 for weak RNG in IoT devices and historical examples)*

Exercise 3: Inspect TCP Stack Fingerprints

(See Slides 036-045 for TCP stack fingerprinting methodology, known-good profiles, and deviation scoring)

The TCP stack fingerprinter compares a device's observed TCP/IP behaviour (initial window size, TTL, TCP options ordering, window scaling, selective acknowledgement support) against the known-good profile for its vendor/firmware combination. Counterfeit devices often run on different hardware or OS versions than genuine devices, producing subtle but detectable TCP stack deviations.

Architecture: What This Exercise Tests

```
graph LR
    subgraph "Stack Fingerprinter (stack_fingerprinter.py)"
        A[TCP SYN/ACK from Phase 2] --> B[Extract TCP parameters]
        B --> C1[initial_window_size]
        B --> C2[ip_ttl]
        B --> C3[tcp_options_order]
        B --> C4[window_scaling]
        B --> C5[selective_ack]
    end
    subgraph "Profile Comparison"
        C1 & C2 & C3 & C4 & C5 --> D[Known-good profile<br/>from
vendor DB]
        D --> E[Deviation score 0-100]
        E --> F{Threshold}
        F -->|score < 15| G[genuine]
        F -->|score > 15| H[stack_deviation indicator]
    end
    subgraph "API"
        H --> I["GET /v1/supply-chain/{scan_id}/counterfeit<br/>
(stack_deviation embedded)"]
        H --> J["GET /v1/supply-chain/{scan_id}/firmware<br/>
(firmware_indicators)"]
    end
```

Step 1: View firmware integrity results (includes stack fingerprint data)

```
# Firmware integrity check includes stack fingerprint indicators
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" | jq .

# Summary: devices checked and mismatches found
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" \
  | jq '{devices_checked: .data.devices_checked,
```

```
        hash_mismatches: .data.hash_mismatches,
        mismatch_rate: (.data.hash_mismatches / .data.devices_checked
       * 100 | round)}'
```

What you should see:

```
{
  "devices_checked": 20,
  "hash_mismatches": 4,
  "mismatch_rate": 20
}
```

Step 2: Examine firmware integrity results per device

```
# Devices with firmware indicators (anomalies)
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[] | select(.firmware_indicators != null and
        (.firmware_indicators | length) > 0) |
       {ip: .device_ip, vendor: .vendor, firmware_version:
        .firmware_version,
        indicators: .firmware_indicators}]'

# All devices with their firmware versions
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[] | {ip: .device_ip, vendor: .vendor,
        firmware_version: .firmware_version,
        has_anomalies: ((.firmware_indicators // []) | length > 0)}]
       | sort_by(.has_anomalies | not)'
```

What you should see for devices with anomalies:

```
[
  {
    "ip": "172.30.0.14",
    "vendor": "Hikvision",
    "firmware_version": "V5.4.5 build 170123",
    "indicators": ["firmware_hash_mismatch", "stack_deviation"]
  },
  {
    "ip": "172.30.0.13",
    "vendor": "Hikvision",
    "firmware_version": "V5.5.800 build 210628",
    "indicators": ["stack_deviation"]
  }
]
```

Step 3: Deep-dive on stack deviation for counterfeit device

```
# Stack deviation details from the counterfeit analysis
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/counterfeit \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[] | select(.device_ip == "172.30.0.14") |
        .analysis_data.counterfeit.indicators[] |
        select(.indicator_type == "stack_deviation") |
        {type: .indicator_type, confidence: .confidence, details:
        .details, severity: .severity}]'

# Compare genuine camera stack to counterfeit camera stack
for IP in 172.30.0.10 172.30.0.14; do
  echo "--- $IP ---"
  curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/counterfeit \
    -H "Authorization: Bearer $TOKEN" \
    | jq --arg ip "$IP" '[.data.results[] | select(.device_ip == $ip)
        |
      {ip: .device_ip, verdict: .analysis_data.counterfeit.verdict,
       stack_indicator: ([.analysis_data.counterfeit.indicators[] |
        select(.indicator_type == "stack_deviation")] | .[0] //
        "none")}]'
done
```

What you should see:

```
--- 172.30.0.10 ---
[{ "ip": "172.30.0.10", "verdict": "genuine", "stack_indicator":
"none" }]
--- 172.30.0.14 ---
[{ "ip": "172.30.0.14", "verdict": "counterfeit",
   "stack_indicator": { "indicator_type": "stack_deviation",
"confidence": 0.7,
     "details": "TCP window size 8192 deviates from Hikvision V5
baseline (65535)", "severity": "medium" } }]
```

> **Troubleshooting:** If `hash_mismatches: 0`, the firmware hash database may be empty or the device firmware versions were not captured during enrichment. Check: `curl -s http://localhost:8100/v1/analytics/behavioral/$SCAN_ID -H "Authorization: Bearer $TOKEN" | jq '[.data.baselines | to_entries[] | .value.firmware_version] | unique'` -- null firmware versions indicate Phase 2 enrichment did not capture banner data.

Questions:

1. Device 172.30.0.14 has a TCP initial window size of 8192, while genuine Hikvision V5 cameras use 65535. How does the TCP initial window size reveal information about the underlying OS? What OS typically uses window size 8192 vs. 65535? *(See Slides 038-040 for TCP window size OS correlation)*
2. TCP stack fingerprinting was originally developed for passive OS detection (p0f, Nmap OS detection). Explain the difference between passive fingerprinting (observing existing traffic) and active fingerprinting (sending probes). Which does

Breakwater use, and what are the legal and operational implications of each in a production environment? *(See Slide 037 for passive vs. active fingerprinting comparison)*

3. A legitimate device might show a TCP stack deviation after a firmware update changes the underlying OS version. Design a procedure for updating the known-good profile database when a vendor releases a new firmware that changes TCP stack behaviour. Who should maintain this database, and how should new profiles be validated? *(See Slides 043-044 for known-good profile lifecycle management)*
4. The `stack_deviation` indicator has `confidence: 0.7`. This is lower than `firmware_hash_mismatch` (0.85) because TCP stack parameters can vary legitimately. List five legitimate causes of TCP window size variation that would not indicate counterfeit: (a) OS version, (b) network tuning, (c) hypervisor, (d) NAT, (e) buffer size tuning. How does the detector avoid flagging these as anomalies? *(See Slide 041 for legitimate TCP variation sources and confidence calibration)*

Exercise 4: Check Firmware Integrity

(See Slides 046-055 for firmware hash verification, vendor DB structure, and integrity checking pipeline)

Firmware integrity checking compares the hash of a device's reported firmware version against a database of known-good firmware hashes published by vendors. A mismatch indicates the firmware may have been modified (tampered with), is a counterfeit build, or is from an unofficial source (grey market firmware). The check uses SHA-256 or MD5 hashes depending on what the vendor publishes.

Architecture: What This Exercise Tests

```
graph LR
    subgraph "Firmware Verifier (firmware_verifier.py)"
        A[Firmware Version String<br/>from Phase 2 banner] --> B[Parse
version tokens]
        B --> C[Firmware Hash DB lookup]
    end
    subgraph "Hash DB (firmware_hash_db.py)"
        D[Vendor firmware releases<br/>+ official hashes] --> C
        C --> E{Hash match?}
        E -->|Match| F[hash_verified: true]
        E -->|No match| G[firmware_hash_mismatch indicator]
        E -->|Not in DB| H[hash_unknown]
    end
    subgraph "Entropy Analyser (entropy_analyzer.py)"
        I[TLS session entropy<br/>from Phase 2] --> J[Shannon entropy]
        J --> K{< 7.0 bits?}
        K -->|Yes| L[weak_rng indicator]
    end
    subgraph "API"
```

```
        F & G & H & L --> M["GET /v1/supply-chain/{scan_id}/firmware"]
    end
```

Step 1: View all firmware integrity results

```
# Complete firmware integrity results
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[] | {ip: .device_ip, vendor: .vendor,
         firmware_version: .firmware_version,
         has_indicators: ((.firmware_indicators // []) | length > 0),
         indicator_count: (.firmware_indicators // [] | length)}]
        | sort_by(.has_indicators | not)'
```

What you should see:

```
[
  { "ip": "172.30.0.14", "vendor": "Hikvision", "firmware_version":
        "V5.4.5 build 170123",
    "has_indicators": true, "indicator_count": 2 },
  { "ip": "172.30.0.13", "vendor": "Hikvision", "firmware_version":
        "V5.5.800 build 210628",
    "has_indicators": true, "indicator_count": 1 },
  { "ip": "172.30.0.10", "vendor": "Hikvision", "firmware_version":
        "V5.5.800 build 210628",
    "has_indicators": false, "indicator_count": 0 },
  { "ip": "172.30.0.30", "vendor": "Synology",  "firmware_version":
        "DSM 7.1-42661",
    "has_indicators": false, "indicator_count": 0 }
]
```

Step 2: Analyse firmware version distribution

```
# Group devices by firmware version (detect fleet-wide version
        lagging)
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results | group_by(.vendor)[] |
        {vendor: .[0].vendor,
         device_count: length,
         firmware_versions: [.[].firmware_version] | unique,
         version_diversity: ([.[].firmware_version] | unique |
        length)}]
        | sort_by(-.version_diversity)'

# Find devices with firmware versions not in hash DB (unknown = not
        verified)
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" \
```

```
  | jq '[.data.results[] | select(.firmware_version == null or
        .firmware_version == "unknown") |
       {ip: .device_ip, vendor: .vendor}]'
```

What you should see for version diversity:

```
[
  { "vendor": "Hikvision", "device_count": 5, "firmware_versions":
        ["V5.4.5 build 170123", "V5.5.800 build 210628"],
       "version_diversity": 2 },
  { "vendor": "Synology",  "device_count": 2, "firmware_versions":
        ["DSM 7.1-42661"], "version_diversity": 1 },
  { "vendor": "QNAP",      "device_count": 2, "firmware_versions":
        ["QTS 5.0.1-2376"], "version_diversity": 1 }
]
```

Step 3: Cross-reference firmware anomalies with counterfeit verdicts

```
# Devices with firmware hash mismatches -- are they also flagged as
       counterfeit?
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[] | select((.firmware_indicators // []) |
       length > 0) | .device_ip]' > /tmp/firmware_anomalies.json

curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/counterfeit \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[] | select(.analysis_data.counterfeit.verdict
       != "genuine") | .device_ip]' > /tmp/counterfeit_flagged.json

echo "Firmware anomaly IPs:"
cat /tmp/firmware_anomalies.json

echo "Counterfeit flagged IPs:"
cat /tmp/counterfeit_flagged.json
```

What you should see:

```
Firmware anomaly IPs:
["172.30.0.14", "172.30.0.13"]

Counterfeit flagged IPs:
["172.30.0.14", "172.30.0.13", "172.30.0.20", "172.30.0.21"]
```

Devices 172.30.0.14 and 172.30.0.13 appear in both lists (firmware anomaly AND counterfeit flagged). Devices 172.30.0.20 and 172.30.0.21 are counterfeit-flagged from MAC/timing anomalies but do not have firmware hash anomalies.

> **Troubleshooting:** If all `firmware_indicators` fields are `null` or empty arrays, the firmware hash DB did not find a match for any device's firmware version. This is expected for devices with generic or unknown firmware strings. The IoT simulation configures specific firmware versions that appear in the hash DB.

Questions:

1. Device 172.30.0.14 has firmware `V5.4.5 build 170123` while genuine cameras use `V5.5.800 build 210628`. The build date (170123 = January 23, 2017) is 4 years older than the other cameras. What supply chain scenario could explain a device running 4-year-old firmware? Consider: (a) grey market sale, (b) cloned firmware, (c) deliberately downgraded. *(See Slides 048-050 for firmware version analysis and grey market indicators)*
2. The firmware hash DB is populated from vendor-published release notes. What are the practical limitations of this approach: which vendors consistently publish firmware hashes, and which do not? How does the absence of vendor hash publication affect the detection rate for this class of counterfeit? *(See Slide 051 for vendor hash publication practices in the IoT ecosystem)*
3. Two Hikvision cameras (172.30.0.10 and 172.30.0.13) run the same firmware `V5.5.800 build 210628`, but only 172.30.0.13 has a `stack_deviation` indicator. What does this tell you: could two genuine cameras with the same firmware produce different TCP stack parameters? What hardware-level factor might explain the difference? *(See Slide 053 for same-firmware stack variation analysis)*
4. The entropy analyser checks for weak RNG by measuring the Shannon entropy of TLS session IDs and nonces. A genuine device's TLS nonces should have entropy close to 8 bits/byte (maximum). Describe what low entropy (below 7.0 bits/byte) looks like in terms of actual byte patterns, and what attack this enables against TLS session establishment. *(See Slides 054-055 for RNG entropy analysis and TLS key prediction attacks)*

Exercise 5: Score Vendor Trust

(See Slides 056-065 for vendor scorecard methodology, 10 scoring dimensions, and grade assignment)

The vendor scorecard assigns a trust grade (A-F) to each manufacturer based on 10 dimensions: patch cadence, CVE response time, SBOM availability, firmware signing, bug bounty programme, security disclosure policy, EOL device support, secure boot support, hardware security features, and regulatory compliance. A vendor's grade directly informs procurement decisions and acceptable use policies.

Architecture: What This Exercise Tests

```
graph LR
    subgraph "Vendor Scorecard (vendor_scorecard.py)"
        A[Vendor DB<br/>vendor_db.py] --> B[VendorScorecard]
        C[CVE History] --> B
        D[Device count<br/>from scan] --> B
    end
    subgraph "10 Dimensions"
        B --> D1[patch_cadence]
        B --> D2[cve_response_time]
        B --> D3[sbom_availability]
```

```
        B --> D4[firmware_signing]
        B --> D5[bug_bounty]
        B --> D6[disclosure_policy]
        B --> D7[eol_support]
        B --> D8[secure_boot]
        B --> D9[hardware_security]
        B --> D10[regulatory_compliance]
    end
    subgraph "Grade Engine"
        D1 & D2 & D3 & D4 & D5 & D6 & D7 & D8 & D9 & D10 -->
E["overall_score = avg(dimensions)"]
        E --> F["A (90+) / B (80+) / C (70+) / D (60+) / F (<60)"]
    end
```

Step 1: Get all vendor scorecards

```
# All vendor scorecards
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" | jq .

# Summary: vendors and their grades
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.vendors[] | {vendor: .vendor, grade: .grade,
         score: .trust_score, device_count, vuln_count}]
        | sort_by(-.score)'
```

What you should see:

```
[
  { "vendor": "Synology",  "grade": "B", "score": 82.0,
        "device_count": 2,
    "vuln_count": 2 },
  { "vendor": "QNAP",      "grade": "C", "score": 71.5,
        "device_count": 2,
    "vuln_count": 4 },
  { "vendor": "Hikvision", "grade": "D", "score": 58.0,
        "device_count": 5,
    "vuln_count": 8 },
  { "vendor": "Dahua",     "grade": "D", "score": 55.0,
        "device_count": 3,
    "vuln_count": 7 }
]
```

Step 2: Inspect dimension scores for a specific vendor

```
# Hikvision vendor summary from the public scan
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.vendors[] | select(.vendor == "Hikvision")] | .[0]'
```

What you should see:

```
{
  "vendor": "Hikvision",
  "grade": "D",
  "score": 58,
  "device_count": 5,
  "vuln_count": 8
}
```

Step 3: Compare vendors across critical dimensions

```
# Side-by-side comparison of public vendor summaries
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.vendors[] | {vendor, grade, score: .trust_score,
        device_count, vuln_count}]
        | sort_by(-.score)'
```

What you should see for firmware_signing:

```
[
  { "vendor": "Synology",  "grade": "B", "dim_score": 95.0,
        "dim_grade": "A" },
  { "vendor": "QNAP",      "grade": "C", "dim_score": 90.0,
        "dim_grade": "A" },
  { "vendor": "Hikvision", "grade": "D", "dim_score": 80.0,
        "dim_grade": "B" },
  { "vendor": "Dahua",     "grade": "D", "dim_score": 75.0,
        "dim_grade": "C" }
]
```

> **Troubleshooting:** If `vendor_count: 0`, no vendor scorecards were generated. Vendor scoring requires device identification (vendor field populated). Verify: `curl -s http://localhost:8100/v1/analytics/graph/$SCAN_ID -H "Authorization: Bearer $TOKEN" | jq '[.data.nodes[] | select(.node_type == "device") | .metadata.vendor] | unique'` -- if all vendors are `null`, Phase 1 identification did not run.

Questions:

1. Hikvision scores "F" on `sbom_availability` (30 points) and `bug_bounty` (40 points). Explain the security significance of each: (a) why does not publishing an SBOM make a device harder to secure, and (b) why does a bug bounty programme improve the security posture of a vendor's products? *(See Slides 059-060 for SBOM availability and bug bounty scoring rationale)*
2. The for Hikvision is "Restricted use -- segment from critical infrastructure." Based on the dimension scores, write a specific network policy: which network zones should Hikvision cameras be excluded from, and what monitoring controls should be applied to those deployed in acceptable zones? *(See Slides 063-064 for procurement policy s from vendor scores)*

3. Synology scores "B" overall while Dahua scores "D" despite both making NAS/camera products. Compare their dimension profiles and identify the specific dimensions where Synology most outperforms Dahua. What specific vendor practices produce those higher scores? *(See Slide 062 for comparative vendor analysis)*
4. The vendor scorecard is based on publicly available information (published CVE response times, SBOM programmes, bug bounty policies). How would you verify these scores in practice? For each of the 10 dimensions, propose one verifiable data source that an organisation could use to independently validate the score. *(See Slide 065 for vendor score validation methodology)*

Exercise 6: Assess EU CRA Compliance

(See Slides 066-075 for EU Cyber Resilience Act requirements, 13-point checklist, and compliance scoring)

The EU Cyber Resilience Act (CRA) entered into force in December 2024 and requires IoT products sold in the EU to meet 13 essential cybersecurity requirements. Breakwater assesses each scanned device against these requirements and produces a compliance report that identifies which requirements pass, fail, or partially pass.

Architecture: What This Exercise Tests

```
graph LR
    subgraph "CRA Compliance (cra_compliance.py)"
        A[Device scan data] --> B[CRAComplianceChecker]
        B --> C1["CRA-01: No known exploitable CVEs at shipment"]
        B --> C2["CRA-02: Unique default credentials"]
        B --> C3["CRA-03: Secure update mechanism"]
        B --> C4["CRA-04: Minimal attack surface"]
        B --> C5["CRA-05: Access control"]
        B --> C6["CRA-06: Data protection"]
        B --> C7["CRA-07: Security communications"]
        B --> C8["CRA-08: Limited data collection"]
        B --> C9["CRA-09: Resilience"]
        B --> C10["CRA-10: Availability assurance"]
        B --> C11["CRA-11: SBOM provision"]
        B --> C12["CRA-12: Vulnerability disclosure"]
        B --> C13["CRA-13: Incident reporting"]
    end
    subgraph "API"
        C1 & C2 & C3 --> D["GET /v1/supply-
chain/{scan_id}/compliance"]
    end
```

Step 1: Get fleet-level CRA compliance summary

```
# Fleet compliance overview
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/compliance \
  -H "Authorization: Bearer $TOKEN" \
  | jq '{device_count: .data.device_count,
         fleet_compliance_pct: .data.fleet_compliance_pct,
         results: [.data.results[] | {ip: .device_ip, vendor: .vendor,
           compliance_pct, passed, failed, partial}]
           | sort_by(-.compliance_pct)}'
```

What you should see:

```
{
  "device_count": 20,
  "fleet_compliance_pct": 47.3,
  "results": [
    { "ip": "172.30.0.32", "vendor": "Cisco", "compliance_pct": 76.9,
      "passed": 10, "failed": 2, "partial": 1 },
    { "ip": "172.30.0.30", "vendor": "Synology", "compliance_pct":
      69.2, "passed": 9, "failed": 3, "partial": 1 },
    { "ip": "172.30.0.10", "vendor": "Hikvision", "compliance_pct":
      38.5, "passed": 5, "failed": 7, "partial": 1 },
    { "ip": "172.30.0.40", "vendor": "Siemens", "compliance_pct":
      30.8, "passed": 4, "failed": 8, "partial": 1 }
  ]
}
```

Step 2: Inspect per-device CRA requirement detail

```
# Full CRA assessment for the camera (Hikvision)
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/compliance \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[] | select(.device_ip == "172.30.0.10") |
        {ip: .device_ip, vendor: .vendor, compliance_pct,
         requirements: [.requirements[] | {req_id, title, status,
        evidence, remediation}]}] | .[0]'
```

What you should see:

```
{
  "ip": "172.30.0.10",
  "vendor": "Hikvision",
  "compliance_pct": 38.5,
  "requirements": [
    { "req_id": "CRA-01", "title": "No known exploitable CVEs at
      shipment",
      "status": "fail", "evidence": "12 known CVEs detected including
        CVE-2021-36260 (CVSS 9.8)",
      "remediation": "Apply latest firmware update to address known
        CVEs" },
    { "req_id": "CRA-02", "title": "Unique default credentials",
      "status": "fail", "evidence": "Default credentials admin/12345
        shared across device line",
```

```
      "remediation": "Enforce unique per-device credentials at
        manufacturing" },
    { "req_id": "CRA-03", "title": "Secure update mechanism",
      "status": "partial", "evidence": "OTA update available but
        signature verification is optional",
      "remediation": "Enable mandatory firmware signature
        verification" },
    { "req_id": "CRA-11", "title": "SBOM provision",
      "status": "fail", "evidence": "No SBOM provided with device or
        available from vendor",
      "remediation": "Request SBOM from Hikvision or generate via
        firmware extraction" }
  ]
}
```

Step 3: Identify common CRA failures across the fleet

```
# Which CRA requirements fail most frequently?
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/compliance \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[].requirements[] | select(.status == "fail") |
        .req_id]
        | group_by(.) | map({requirement: .[0], fail_count: length})
        | sort_by(-.fail_count)'
```

What you should see:

```
[
  { "requirement": "CRA-02", "fail_count": 14 },
  { "requirement": "CRA-01", "fail_count": 12 },
  { "requirement": "CRA-11", "fail_count": 11 },
  { "requirement": "CRA-03", "fail_count": 8  },
  { "requirement": "CRA-05", "fail_count": 7  }
]
```

CRA-02 (unique default credentials) fails for 14 of 20 devices -- the single most common compliance gap across the IoT fleet.

> **Troubleshooting:** If `fleet_compliance_pct: 0.0`, no CRA assessments were generated. The CRA checker runs per device and requires device identification data. Verify with: `curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/compliance -H "Authorization: Bearer $TOKEN" | jq '.data.device_count'` -- if zero, the supply chain phase did not persist CRA results.

Questions:

1. The fleet's average CRA compliance is 47.3% -- less than half of the 13 requirements are met on average. CRA non-compliance can result in fines up to €15 million or 2.5% of global turnover for manufacturers. From the fleet operator's (not manufacturer's) perspective, what legal obligations does this non-compliance create? *(See Slide 068 for CRA obligations for device operators vs. manufacturers)*

2. CRA-02 (unique default credentials) fails for 14 of 20 devices. This single requirement, if fixed, would address the most common MITRE ATT&CK technique (T0812 Default Credentials) identified in Phase 4. Calculate the compounded risk: if 14 devices are CRA-02 non-compliant AND each was confirmed exploitable in Phase 5, what is the total attack surface? *(See Slide 070 for CRA-MITRE ATT&CK mapping)*
3. CRA-11 (SBOM provision) fails for 11 devices. The CRA requires manufacturers to make SBOMs available to market surveillance authorities. If a manufacturer provides an SBOM in CycloneDX format but it is missing transitive dependencies, does the device pass CRA-11 "partial" or "fail"? Justify your answer using the CRA essential requirements text. *(See Slides 071-072 for CRA-11 SBOM provision requirements and evaluation criteria)*
4. Write a CRA compliance remediation roadmap for Hikvision cameras (compliance_pct: 38.5%). Order the 8 failing/partial requirements by: (1) immediate operator action (things the operator can do without vendor cooperation), (2) vendor-dependent actions (requiring firmware update), (3) procurement replacement (requiring device replacement). *(See Slides 073-075 for CRA remediation prioritisation framework)*

Exercise 7: Detect Abandoned Components

(See Slides 076-082 for component health classification, abandoned component detection, and EOL risk scoring)

Abandoned components are open-source libraries or firmware modules that are no longer actively maintained -- no new releases, no bug fixes, no security patches for 2+ years. Devices depending on abandoned components accumulate CVEs indefinitely because no vendor will release patches. The component health classifier assigns each dependency to one of four categories: `healthy`, `degraded`, `abandoned`, or `compromised`.

Architecture: What This Exercise Tests

```
graph LR
    subgraph "Component Health (component_health.py)"
        A[SBOM components] --> B[Health Classifier]
        B --> C1{Last release date}
        C1 -->|< 6 months| D1[healthy]
        C1 -->|6–24 months| D2[degraded]
        C1 -->|> 24 months| D3[abandoned]
        B --> C2{Active CVEs?}
        C2 -->|Yes + abandoned| D4[compromised]
    end
    subgraph "Scoring"
        D1 & D2 & D3 & D4 --> E[criticality_score 0–1]
        E --> F[component blast_radius<br/>from dep_graph]
    end
    subgraph "API"
```

```
        F --> G["GET /v1/supply-chain/{scan_id}<br/>(component health
embedded in sbom_data)"]
    end
```

Step 1: Find abandoned components in SBOM data

```
# Access SBOM component data including health classification
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.analyses[] | select(.sbom_data.component_count > 0) |
        {ip: .device_ip,
         total_components: .sbom_data.component_count,
         blast_radius: .sbom_data.blast_radius}]
        | sort_by(-.blast_radius)'

# Summary of SBOM coverage
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '{total_components: .data.total_components,
         total_sbom_vulnerabilities: .data.total_sbom_vulnerabilities,
         vuln_per_component: (.data.total_sbom_vulnerabilities /
        .data.total_components * 100 | round)}'
```

What you should see:

```
[
  { "ip": "172.30.0.30", "total_components": 48, "blast_radius": 48 },
  { "ip": "172.30.0.31", "total_components": 45, "blast_radius": 45 },
  { "ip": "172.30.0.10", "total_components": 22, "blast_radius": 22 }
]

{ "total_components": 187, "total_sbom_vulnerabilities": 43,
        "vuln_per_component": 23 }
```

23 vulnerabilities per 100 components -- a typical rate for IoT firmware that bundles many C libraries without active patching.

Step 2: Find devices with highest transitive CVE exposure

```
# Rank devices by transitive CVE count (vulnerabilities in indirect
        dependencies)
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.analyses[] | select(.sbom_data.component_count > 0) |
        {ip: .device_ip,
         total_components: .sbom_data.component_count,
         sbom_vulnerabilities: .sbom_vulnerabilities,
         vuln_density: (.sbom_vulnerabilities /
        .sbom_data.component_count * 100 | round)}]
        | sort_by(-.sbom_vulnerabilities)'
```

What you should see:

```
[
  { "ip": "172.30.0.30", "total_components": 48,
        "sbom_vulnerabilities": 15, "vuln_density": 31 },
  { "ip": "172.30.0.31", "total_components": 45,
        "sbom_vulnerabilities": 13, "vuln_density": 29 },
  { "ip": "172.30.0.10", "total_components": 22,
        "sbom_vulnerabilities": 7,  "vuln_density": 32 }
]
```

Step 3: Look at transitive CVE examples

```
# Get transitive CVE examples from the highest-risk device
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.analyses[] | select(.device_ip == "172.30.0.30") |
        .sbom_data | {
          component_count,
          diamond_deps_count: (.diamond_deps | length),
          transitive_cve_examples: .transitive_cves[:3]
        }] | .[0]'
```

What you should see:

```
{
  "component_count": 48,
  "diamond_deps_count": 3,
  "transitive_cve_examples": [
    { "cve_id": "CVE-2022-0778", "severity": "high",
      "cpe": "cpe:2.3:a:openssl:openssl:1.1.1t:*:*:*:*:*:*:*",
      "component_path": ["Synology DSM", "apache2", "mod_ssl",
        "openssl"] },
    { "cve_id": "CVE-2023-0286", "severity": "high",
      "cpe": "cpe:2.3:a:openssl:openssl:1.1.1t:*:*:*:*:*:*:*",
      "component_path": ["Synology DSM", "apache2", "mod_ssl",
        "openssl"] },
    { "cve_id": "CVE-2021-22945", "severity": "critical",
      "cpe": "cpe:2.3:a:haxx:libcurl:7.87.0:*:*:*:*:*:*:*",
      "component_path": ["Synology DSM", "wget", "libcurl"] }
  ]
}
```

The `component_path` shows the chain from the firmware's top-level software to the vulnerable component: `Synology DSM -> apache2 -> mod_ssl -> openssl`. This path is 4 hops deep and would be invisible without SBOM analysis.

> **Troubleshooting:** If `transitive_cves` is an empty list, the transitive CVE lookup did not find matches for the components in the hash DB. This is expected in an offline lab -- the SBOM analyser uses a pre-populated component

> vulnerability cache. Ensure the IoT simulation is running with `docker compose ps` and `./wait-for-lab.sh`.

Questions:

1. The Synology NAS has `component_path: ["Synology DSM", "apache2", "mod_ssl", "openssl"]` for CVE-2022-0778. This CVE was patched in openssl 1.1.1n, but the device uses 1.1.1t (which IS patched). Yet the CVE appears in the transitive CVE list. What does this tell you about the SBOM analyser's false positive rate, and how should it validate CVE applicability against actual version numbers? *(See Slide 079 for transitive CVE applicability validation)*
2. The NAS has a `vuln_density` of 31 CVEs per 100 components. Compare this to the camera's 32 CVEs per 100 components -- they are similar despite very different device types. What does similar vuln density across different device types suggest about the underlying cause? Hint: both devices ship C libraries (libc, openssl, libcurl) with known CVEs. *(See Slide 080 for cross-device vuln density analysis)*
3. If a component is `abandoned` (no releases in 24+ months) AND has active CVEs, it is classified as `compromised`. What procurement policy should an organisation adopt for devices with `compromised` components: (a) immediate replacement, (b) compensating controls, (c) monitor and accept? Justify your choice based on the availability of patches. *(See Slides 080-082 for abandoned component risk policy)*
4. SBOM adoption in IoT is still low -- only 60% of devices in this scan had SBOM data. The US Executive Order 14028 (May 2021) requires SBOMs for software sold to the federal government, and the EU CRA requires SBOMs for products sold in the EU. Design an SLA clause for an IoT device procurement contract that specifies SBOM obligations: format (SPDX or CycloneDX), depth (direct only vs. transitive), update frequency, and availability to operators. *(See Slide 082 for SBOM contractual requirements)*

Exercise 8: Run Phylogenetic Analysis

(See Slides 083-090 for firmware phylogenetics, similarity scoring, and evolutionary tree construction)

Firmware phylogenetic analysis constructs an evolutionary tree of firmware versions by computing pairwise similarity scores between firmware images. Devices sharing a common firmware ancestor (same code base, forked versions, or OEM relationships) cluster together in the tree. This reveals hidden vendor relationships, detects firmware re-use across supposedly independent products, and identifies when a vulnerability in one firmware lineage affects all related devices.

Architecture: What This Exercise Tests

```
graph LR
    subgraph "Phylogenetic Engine (phylogenetic.py)"
        A[Firmware version strings<br/>per device] --> B[Pairwise
similarity<br/>Levenshtein + hash distance]
        B --> C[Distance matrix]
```

```
        C --> D["UPGMA clustering<br/>(hierarchical)"]
        D --> E[PhylogeneticNode tree]
    end
    subgraph "API"
        E --> F["GET /v1/supply-chain/{scan_id}/firmware<br/>
(phylogenetic data embedded)"]
    end
```

Step 1: View firmware versions for phylogenetic grouping

```
# Get all firmware versions to understand the lineage landscape
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results | group_by(.vendor)[] | {
      vendor: .[0].vendor,
      firmware_lineage: [.[].firmware_version] | unique | sort,
      device_count: length
    }]'
```

What you should see:

```
[
  {
    "vendor": "Hikvision",
    "firmware_lineage": ["V5.4.5 build 170123", "V5.5.800 build
        210628"],
    "device_count": 5
  },
  {
    "vendor": "Synology",
    "firmware_lineage": ["DSM 7.1-42661"],
    "device_count": 2
  },
  {
    "vendor": "QNAP",
    "firmware_lineage": ["QTS 5.0.1-2376"],
    "device_count": 2
  }
]
```

Step 2: Analyse firmware lineage for vulnerability propagation

```
# Devices sharing the same firmware version share the same CVE surface
# Find firmware versions used by multiple devices
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results | group_by(.firmware_version)[] | select(length
        > 1) |
        {firmware_version: .[0].firmware_version,
         device_count: length,
```

```
         device_ips: [.[].device_ip],
         vendors: [.[].vendor] | unique}]
        | sort_by(-.device_count)'

# Cross-reference shared firmware with CVE count
SHARED_FW="V5.5.800 build 210628"
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" \
  | jq --arg fw "$SHARED_FW" '[.data.results[] |
       select(.firmware_version == $fw) |
     {ip: .device_ip, vendor: .vendor, firmware_version}]'
```

What you should see:

```
[
  {
    "firmware_version": "V5.5.800 build 210628",
    "device_count": 4,
    "device_ips": ["172.30.0.10", "172.30.0.11", "172.30.0.12",
        "172.30.0.13"],
    "vendors": ["Hikvision"]
  }
]
```

Step 3: Interpret phylogenetic implications for blast radius

```
# If CVE-2021-36260 affects Hikvision V5.x, how many devices are in
       the blast radius?
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[] | select(.vendor == "Hikvision") |
       {ip: .device_ip, firmware_version}]
       | {hikvision_count: length, devices: .}'

# Build a vulnerability blast radius: CVE affects firmware lineage
       V5.x
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[] | select(.vendor == "Hikvision" and
        (.firmware_version // "" | startswith("V5"))) | .device_ip]
       | {affected_devices: ., count: length}'
```

What you should see:

```
{
  "affected_devices": ["172.30.0.10", "172.30.0.11", "172.30.0.12",
       "172.30.0.13", "172.30.0.14"],
  "count": 5
}
```

All 5 Hikvision cameras running V5.x firmware are in the blast radius of CVE-2021-36260 -- a phylogenetic vulnerability that propagates through the entire V5 lineage.

> **Troubleshooting:** If no devices share firmware versions, each device was identified with a unique firmware string (no lineage overlap). This can happen with heterogeneous networks. The IoT simulation pre-configures camera devices to share firmware versions for this exercise.

Questions:

1. Four cameras share firmware `V5.5.800 build 210628`. This means a CVE affecting this firmware version has a blast radius of 4 devices automatically -- before any network-level lateral movement. Compare this to the Phase 4 blast radius (which required network traversal). How does supply chain blast radius differ conceptually from network blast radius? *(See Slides 085-086 for supply chain vs. network blast radius)*
2. The counterfeit camera (172.30.0.14) runs `V5.4.5 build 170123` -- an older firmware branch than the other cameras. Using phylogenetic thinking, is the V5.4.5 lineage a parent or child of the V5.5.800 lineage? What CVEs exist in V5.4.5 that were patched in V5.5.800? *(See Slide 088 for firmware lineage direction and CVE inheritance)*
3. OEM relationships in IoT are common: Camera vendor A licenses firmware from a third party, and Camera vendor B independently licenses the same firmware. How would a phylogenetic analysis detect this shared ancestry even when the product names are different? What security implication arises when two "competing" products share a firmware ancestor? *(See Slide 089 for OEM firmware lineage detection)*
4. Phylogenetic analysis of firmware versions is analogous to biological phylogenetics (evolutionary biology). Both construct trees from similarity distances. Design a test to validate the phylogenetic tree's accuracy: what ground truth data would you use, and how would you measure reconstruction error? *(See Slide 090 for phylogenetic tree validation methodology)*

Exercise 9: Simulate the Attack Blast Radius

(See Slides 091-098 for Monte Carlo supply chain attack simulation, propagation probability model, and blast radius calculation)

The Monte Carlo simulation models a supply chain attack: an adversary compromises one component in the dependency graph (e.g., a widely-used library), and the simulation propagates the compromise through the dependency tree using probabilistic infection. After N=1000 simulations, the result is a probability distribution over blast radius (how many dependent components are compromised).

Architecture: What This Exercise Tests

```
graph LR
    subgraph "Attack Simulation (attack_sim.py)"
```

```
        A[Dependency Graph<br/>from all SBOMs] --> B[Monte Carlo
Sampler]
        C[target_component] --> B
        D[propagation_prob<br/>default 0.85] --> B
        B --> E[N simulations]
        E --> F[BFS propagation<br/>per simulation]
        F --> G[blast_radius distribution]
    end
    subgraph "AttackSimResult"
        G --> H[blast_radius: median]
        G --> I[probability: P(any device affected)]
        G --> J[critical_paths: top propagation routes]
        G --> K[affected_devices list]
    end
    subgraph "API"
        H & I & J & K --> L["POST /v1/supply-
chain/{scan_id}/simulate"]
    end
```

Step 1: Simulate a compromise of a common library

```
# First, find component names from the SBOM
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.analyses[] | .sbom_data.components // [] |
        .[] | if type == "object" then .name else . end] | unique | .
        [:10]'
```

Step 2: Run the Monte Carlo simulation when SBOM components are present

```
# The public simulation may report zero SBOM components for some
        scans. In that case,
# record the SBOM coverage gap and skip the Monte Carlo POST for this
        scan.
COMPONENT=$(curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq -r '[.data.analyses[] | .sbom_data.components // [] |
        .[] | if type == "object" then .name else . end] | unique | .
        [0] // empty')

if [ -z "$COMPONENT" ]; then
  echo "No SBOM components found in this public scan; document the
        SBOM coverage gap."
else
  curl -s -X POST http://localhost:8100/v1/supply-
        chain/$SCAN_ID/simulate \
    -H "Authorization: Bearer $TOKEN" \
    -H "Content-Type: application/json" \
    -d "{\"target_component\": \"$COMPONENT\", \"n_simulations\":
        1000, \"propagation_prob\": 0.85}" | jq .
fi
```

```
# Key blast radius metrics
if [ -n "$COMPONENT" ]; then
  curl -s -X POST http://localhost:8100/v1/supply-
        chain/$SCAN_ID/simulate \
    -H "Authorization: Bearer $TOKEN" \
    -H "Content-Type: application/json" \
    -d "{\"target_component\": \"$COMPONENT\", \"n_simulations\":
        1000, \"propagation_prob\": 0.85}" \
    | jq '{compromised_component: .data.compromised_component,
           blast_radius: .data.blast_radius,
           probability: .data.probability,
           affected_devices_count: (.data.affected_devices | length),
           critical_path_count: (.data.critical_paths | length)}'
fi
```

What you should see:

```
{
  "compromised_component": "openssl",
  "blast_radius": 18,
  "probability": 0.94,
  "affected_devices_count": 7,
  "critical_path_count": 3
}
```

Step 3: Compare blast radius for different propagation probabilities

```
if [ -n "$COMPONENT" ]; then
  for PROB in 0.5 0.85 0.99; do
    echo "--- propagation_prob=$PROB ---"
    curl -s -X POST http://localhost:8100/v1/supply-
        chain/$SCAN_ID/simulate \
      -H "Authorization: Bearer $TOKEN" \
      -H "Content-Type: application/json" \
      -d "{\"target_component\": \"$COMPONENT\", \"n_simulations\":
        500, \"propagation_prob\": $PROB}" \
      | jq '{prob: .data.probability, blast_radius:
        .data.blast_radius,
             affected_devices: (.data.affected_devices | length)}'
  done
fi
```

What you should see:

```
--- propagation_prob=0.5 ---
{ "prob": 0.72, "blast_radius": 8, "affected_devices": 4 }
--- propagation_prob=0.85 ---
{ "prob": 0.94, "blast_radius": 18, "affected_devices": 7 }
--- propagation_prob=0.99 ---
{ "prob": 0.99, "blast_radius": 27, "affected_devices": 9 }
```

Step 4: View critical propagation paths

```
# Get critical paths from the simulation
if [ -n "$COMPONENT" ]; then
  curl -s -X POST http://localhost:8100/v1/supply-
        chain/$SCAN_ID/simulate \
    -H "Authorization: Bearer $TOKEN" \
    -H "Content-Type: application/json" \
    -d "{\"target_component\": \"$COMPONENT\", \"n_simulations\":
        1000, \"propagation_prob\": 0.85}" \
    | jq '.data.critical_paths[:3]'
fi
```

What you should see:

```
[
  ["openssl", "mod_ssl", "apache2", "Synology DSM"],
  ["openssl", "libssl", "wget", "QNAP QTS"],
  ["openssl", "libcrypto", "curl", "camera-firmware"]
]
```

Each critical path shows how an openssl compromise propagates up the dependency tree to reach firmware-level impact.

> **Troubleshooting:** If the component list is empty, the correct public-lab conclusion is not "the endpoint is broken"; it is that the current scan contains no SBOM component graph for Monte Carlo propagation. Record that as a supply-chain visibility gap and continue with counterfeit, firmware, CRA, and vendor-summary analysis.

Questions:

1. With `propagation_prob: 0.85`, compromising openssl has `probability: 0.94` of affecting at least one device and a median `blast_radius: 18` components. The probability is not 1.0 because some dependency edges may not propagate the compromise (e.g., the library is present but the vulnerable code path is not exercised). What real-world factors determine whether a compromised component actually affects a dependent device? *(See Slides 093-094 for propagation probability calibration)*
2. The critical paths show openssl propagating through `mod_ssl -> apache2 -> Synology DSM`. This is a 4-hop dependency chain. In a real supply chain attack (e.g., XZ Utils backdoor, March 2024), how did attackers exploit a similar deep-dependency position to affect millions of systems? Draw the parallels to the simulation. *(See Slides 095-096 for real-world supply chain attack case studies)*
3. Increasing `propagation_prob` from 0.5 to 0.99 more than doubles the blast radius (8 to 27) and increases affected devices from 4 to 9. The propagation probability is meant to model the likelihood that a dependency relationship results in actual compromise. What factors would you use to calibrate this value for a real deployment -- consider: code coverage, API surface, isolation boundaries. *(See Slide 097 for propagation probability calibration methodology)*
4. The Monte Carlo approach runs 1000 simulations. How would you validate that 1000 simulations is sufficient for accurate blast radius estimation? What is the standard

error of the estimate, and what number of simulations would reduce it to ±1 component? *(See Slide 098 for Monte Carlo convergence analysis)*

Exercise 10: Export the Supply Chain Report

(See Slides 099-110 for supply chain report structure, risk summary, and executive presentation)

The supply chain report aggregates all Phase 9 findings into a comprehensive document: SBOM coverage, counterfeit device findings, vendor trust grades, CRA compliance gaps, firmware integrity results, and simulation blast radius estimates. This report is designed to support procurement decisions, regulatory filings, and board-level risk briefings.

Architecture: What This Exercise Tests

```
graph LR
    subgraph "Report Aggregation"
        A["GET /v1/supply-chain/{scan_id}"] --> E[Report Builder]
        B["GET /v1/supply-chain/{scan_id}/counterfeit"] --> E
        C["GET /v1/supply-chain/{scan_id}/vendors"] --> E
        D["GET /v1/supply-chain/{scan_id}/compliance"] --> E
        F["GET /v1/supply-chain/{scan_id}/firmware"] --> E
    end
    subgraph "Report Sections"
        E --> G1[Executive Summary]
        E --> G2[SBOM Coverage Analysis]
        E --> G3[Counterfeit Device Report]
        E --> G4[Vendor Trust Assessment]
        E --> G5[CRA Compliance Gap Report]
        E --> G6[Firmware Integrity Summary]
    end
```

Step 1: Collect all Phase 9 data for the report

```
# Section 1: SBOM coverage
echo "=== SBOM COVERAGE ==="
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '{hosts_analysed: .data.hosts_analysed,
         sbom_coverage_pct: (.data.devices_with_sbom /
        .data.hosts_analysed * 100 | round),
         total_components: .data.total_components,
         total_sbom_vulnerabilities:
        .data.total_sbom_vulnerabilities}'
```

What you should see:

```
{
  "hosts_analysed": 20,
  "sbom_coverage_pct": 60,
  "total_components": 187,
  "total_sbom_vulnerabilities": 43
}
```

Step 2: Counterfeit and vendor summary

```
# Section 2: Counterfeit findings
echo "=== COUNTERFEIT SUMMARY ==="
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/counterfeit \
  -H "Authorization: Bearer $TOKEN" \
  | jq '{counterfeit_devices: .data.counterfeit,
         suspicious_devices: .data.suspicious,
         pct_flagged: (.data.total_flagged / 20 * 100 | round)}'

# Section 3: Vendor grades
echo "=== VENDOR GRADES ==="
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.vendors[] | {vendor: .vendor, grade: .grade, score:
        .trust_score}]
        | sort_by(-.trust_score)'
```

What you should see:

```
{ "counterfeit_devices": 1, "suspicious_devices": 3, "pct_flagged": 20
        }

[
  { "vendor": "Synology",  "grade": "B", "score": 82.0 },
  { "vendor": "QNAP",      "grade": "C", "score": 71.5 },
  { "vendor": "Hikvision", "grade": "D", "score": 58.0 },
  { "vendor": "Dahua",     "grade": "D", "score": 55.0 }
]
```

Step 3: CRA compliance and firmware summary

```
# Section 4: CRA compliance
echo "=== CRA COMPLIANCE ==="
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/compliance \
  -H "Authorization: Bearer $TOKEN" \
  | jq '{fleet_compliance_pct: .data.fleet_compliance_pct,
         failing_devices: [.data.results[] | select(.compliance_pct <
        50)] | length,
         most_common_gaps: null}'

# Section 5: Firmware integrity
echo "=== FIRMWARE INTEGRITY ==="
curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID/firmware \
```

```
  -H "Authorization: Bearer $TOKEN" \
  | jq '{devices_checked: .data.devices_checked,
         hash_mismatches: .data.hash_mismatches,
         integrity_rate: (((.data.devices_checked -
        .data.hash_mismatches) / .data.devices_checked) * 100 |
        round)}'
```

What you should see:

```
{ "fleet_compliance_pct": 47.3, "failing_devices": 9 }

{ "devices_checked": 20, "hash_mismatches": 4, "integrity_rate": 80 }
```

Step 4: Assemble the executive risk summary

```
# Full executive summary combining all sections
echo "============================================="
echo "PHASE 9 SUPPLY CHAIN RISK EXECUTIVE SUMMARY"
echo "============================================="
echo ""

# Risk score: weighted combination of all findings
SBOM_SCORE=$(curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID \
  -H "Authorization: Bearer $TOKEN" \
  | jq '(.data.devices_with_sbom / .data.hosts_analysed * 100 |
        round)')

INTEGRITY_SCORE=$(curl -s http://localhost:8100/v1/supply-
        chain/$SCAN_ID/firmware \
  -H "Authorization: Bearer $TOKEN" \
  | jq '(((.data.devices_checked - .data.hash_mismatches) /
        .data.devices_checked) * 100 | round)')

CRA_SCORE=$(curl -s http://localhost:8100/v1/supply-
        chain/$SCAN_ID/compliance \
  -H "Authorization: Bearer $TOKEN" \
  | jq '.data.fleet_compliance_pct | round')

echo "SBOM Coverage:        $SBOM_SCORE%"
echo "Firmware Integrity:   $INTEGRITY_SCORE%"
echo "CRA Compliance:       $CRA_SCORE%"
echo ""
echo "Composite Supply Chain Risk Score: $(echo "($SBOM_SCORE +
        $INTEGRITY_SCORE + $CRA_SCORE) / 3" | bc)%"
```

What you should see:

```
SBOM Coverage:        60%
Firmware Integrity:   80%
CRA Compliance:       47%
```

```
Composite Supply Chain Risk Score: 62%
```

A composite score of 62% means the fleet is at moderate-to-high supply chain risk: significant CRA compliance gaps, one confirmed counterfeit device, and 40% of devices without SBOM coverage.

> **Troubleshooting:** If any section returns error codes, collect results individually and note which sections failed. A partial report (e.g., SBOM + firmware only, missing CRA) is still valuable. Present available data with clear notes about missing sections.

Questions:

1. The composite supply chain risk score is 62%. Which of the three components (SBOM coverage, firmware integrity, CRA compliance) is the largest drag on the score? If you could improve only one component by 20 percentage points, which would you choose for maximum risk reduction? *(See Slides 101-104 for composite risk scoring methodology)*
2. One device was classified as `counterfeit` with high confidence (score 72). What are the immediate response actions when a counterfeit device is discovered in a production network? Consider: (a) device isolation, (b) forensic evidence collection, (c) vendor notification, (d) law enforcement, (e) fleet-wide inspection. Prioritise these actions with timelines. *(See Slides 105-107 for counterfeit device incident response playbook)*
3. CRA compliance is 47.3% -- the EU CRA requires 100% by December 2027 (existing products) or by shipment date (new products). Draft a 3-year compliance roadmap with specific milestones: Year 1 (quick wins), Year 2 (vendor-dependent upgrades), Year 3 (replacement of non-compliant devices). Reference specific CRA requirements from your Exercise 6 analysis. *(See Slide 108 for CRA compliance roadmap template)*
4. Review all 10 exercises in this lab. A CISO must present Phase 9 findings to the board in 5 minutes. Write a 5-bullet executive summary that: (a) quantifies supply chain risk in business terms, (b) identifies the highest-priority finding, (c) states what regulatory consequences are at risk, (d) proposes the first remediation action, (e) states the total cost and risk reduction of that action. Use numbers from your exercises throughout. *(See Slides 109-110 for board-level supply chain risk communication)*

Cleanup

```bash
# Remove temporary files created in Exercise 4
rm -f /tmp/firmware_anomalies.json /tmp/counterfeit_flagged.json

# Clear environment variables
unset TOKEN SCAN_ID SHARED_FW SBOM_SCORE INTEGRITY_SCORE CRA_SCORE
```

If you are finished with the lab environment:

```
docker compose down -v    # Stop containers and remove data
```

Troubleshooting Reference

Authentication and Setup

Symptom	Cause	Fix
`Token: null` after login	User account does not exist	Create account via the registration endpoint or reset the lab database
`Scan ID: null`	No completed scans	Run a scan: `python scan_report.py 172.30.0.0/24`
`401 Unauthorized` on any request	Token expired (30 min TTL)	Re-login with the setup block at the top of this lab

Phase 9 API Issues

Symptom	Cause	Fix
`devices_with_sbom: 0`	SBOM extractor found no component data	Verify firmware banners were captured; check `behavioral` endpoint for firmware_version fields
`total_flagged: 0` on counterfeit	No anomalies detected	IoT sim device 172.30.0.14 must be running; verify with `docker compose ps` and `./wait-for-lab.sh`
`vendor_count: 0`	No device identification	Verify Phase 1 identified vendors: `jq '[.data.nodes[] | select(.node_type == "device") | .metadata.vendor] | unique'`
`fleet_compliance_pct: 0.0`	CRA checker returned no results	`device_count: 0` means phase did not persist CRA data; re-run scan
`404` on `/simulate`	Component not in dependency graph	List available components from SBOM data before simulating
`"Aggregation produced no model"`	N/A (Phase 8)	Wrong lab -- this is Phase 9 (supply chain)
`hash_mismatches: 0`	Firmware hash DB empty or no firmware versions	Ensure IoT sim is running with correct firmware version strings

jq Errors

Symptom	Cause	Fix
`Cannot iterate over null`	Field is null	Add fallback: `.sbom_data // {}` or `.firmware_indicators // []`
`Division by zero`	`hosts_analysed: 0`	Guard: `if .data.hosts_analysed > 0 then ... else 0 end`
`string ("openssl") and null cannot be added`	Component name lookup returned null	Use `// ""` null coalescing on string fields

Network and Container Issues

Symptom	Cause	Fix
Connection refused on port 8100	API container not running	`docker compose up -d` and wait 30 seconds
Simulate returns 404	Component not found in combined dep graph	Verify SBOM data exists: `curl -s http://localhost:8100/v1/supply-chain/$SCAN_ID -H "Authorization: Bearer $TOKEN" | jq '.data.total_components'` must be non-zero
IoT sim devices missing	Containers not running	`docker compose up -d` and `./wait-for-lab.sh`

Slide Reference Index

Exercise	Topic	Slides
Setup	Phase 9 overview and supply chain threat model	001-009
1	SBOM analysis and transitive vulnerabilities	010-022 (esp. 013-019)
2	Counterfeit device detection	023-035 (esp. 026-032)
3	TCP stack fingerprinting	036-045 (esp. 038-043)
4	Firmware integrity verification	046-055 (esp. 048-053)
5	Vendor trust scoring (10 dimensions)	056-065 (esp. 059-064)
6	EU CRA compliance assessment	066-075 (esp. 068-074)
7	Abandoned component detection	076-082 (esp. 079-082)
8	Firmware phylogenetic analysis	083-090 (esp. 085-089)
9	Monte Carlo supply chain simulation	091-098 (esp. 093-097)

Exercise	Topic	Slides
10	Supply chain report and executive summary	099-110 (esp. 101-108)
--	API design and endpoints	081-090
--	Dashboard supply chain views	091-100
--	Testing strategy	111-116