SEAS-8414 Week 11 Student Lab Guide

SEAS-8414 Week 11 Student Lab Guide: Formal Protocol Verification

Start Here: Download the Student ZIP

Before running any lab commands, download the Week 11 student package from the course site:

https://8414.bwater.io/downloads/labs/packages/seas8414-blackboard-week-11-2026.05.0_4e76a52f_aws8414.zip

The recommended path is the package Makefile:

```
unzip seas8414-blackboard-week-11-2026.05.0_4e76a52f_aws8414.zip
cd seas8414-blackboard-week-11-2026.05.0_4e76a52f_aws8414
make week11
```

The Makefile calls `run-week11-lab.sh`, extracts the nested runtime ZIP, starts the lab, runs the Week 11 API workflow, saves evidence under `lab-results/week-11/evidence/`, generates `lab-results/week-11/index.html`, and cleans up containers at exit.

You can also download the runner directly from https://8414.bwater.io/downloads/labs/scripts/run-week11-lab.sh.

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

```
unzip seas8414-blackboard-week-11-2026.05.0_4e76a52f_aws8414.zip
cd seas8414-blackboard-week-11-2026.05.0_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/phase11-lab-screencast.mp4
- LLM Prompt: https://8414.bwater.io/downloads/labs/prompts/phase11-lab-llm-prompt.md
- Run Script: https://8414.bwater.io/downloads/labs/scripts/run-week11-lab.sh

---

Breakwater Phase 11: Formal Protocol Verification Lab

Phase 11 introduces machine-checked formal verification of security properties: automated theorem proving over cryptographic protocol models, ProVerif/Tamarin-based counterexample generation, RFC conformance scoring with grammar rule coverage, and pi-calculus process models for IoT protocols. These capabilities move Breakwater from testing actual protocol behavior (Phase 3) to proving mathematical guarantees about protocol security -- or generating concrete attack traces when guarantees cannot be established.

(See Slides 001-009 for Phase 11 overview, formal methods foundations, and verification architecture)

Prerequisites

- Completed Phase 1 lab (scan data with TLS and protocol data)
- Completed Phase 3 lab (protocol transcripts and grammar inference)
- A completed scan with `$SCAN_ID` and `$TOKEN` variables set (see Phase 1 Exercise 6)
- Exposure to formal logic concepts (predicates, propositions, proofs) -- helpful but not required
- No experience with ProVerif or Tamarin required

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`, run `python scan_report.py 172.30.0.0/24` to create a scan.

Phase 11 API Cheatsheet

Endpoint	Method	Description
`/v1/formal/{scan_id}`	GET	Formal verification summary populated by the scan
`/v1/formal/{scan_id}/{ip}`	GET	Per-device verification results
`/v1/formal/results?scan_id={scan_id}`	GET	Dashboard-compatible property summary
`/v1/formal/traces?scan_id={scan_id}`	GET	Dashboard-compatible attack traces
`/v1/formal/conformance?scan_id={scan_id}`	GET	Dashboard-compatible RFC conformance summary

All Phase 11 endpoints require Bearer token authentication.

The public student runtime precomputes Phase 11 formal evidence during the scan. Use the read-only endpoints above. Do not use `POST /v1/formal/{scan_id}/verify` in the public lab; that endpoint is reserved for instructor or development environments with the full formal verification toolchain enabled.

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

Exercise 1: Verify TLS Security Properties

(See Slides 010-025 for TLS formal model, Dolev-Yao adversary, and ProVerif process algebra)

Formal verification proves security properties by exhaustively checking all possible protocol executions, including those with a Dolev-Yao adversary who can intercept, replay, forge, and inject any message in the network. If the proof succeeds, the property holds for all possible executions. If the proof fails, the verifier generates a concrete attack trace showing exactly how the property can be violated.

Architecture: What This Exercise Tests

```
graph LR
    subgraph "Protocol Data (Phase 3)"
        A[TLS Transcripts] --> B[ModelBuilder]
        B --> C[PiCalculusProcess]
    end
    subgraph "Formal Verifier (formal_verify/)"
        C --> D[ProVerifBackend]
        D --> E[PropertyChecker]
        E --> F{Proof or<br/>Counterexample?}
    end
    subgraph "API Layer (formal_router.py)"
        F --> G["GET /v1/formal/{id}"]
    end
    G --> H[VerificationResult JSON]
```

Step 1: Read the scan's formal verification summary

```
# Read comprehensive formal verification evidence populated by the
      scan
curl -s "http://localhost:8100/v1/formal/$SCAN_ID" \
  -H "Authorization: Bearer $TOKEN" | jq '.data.verdicts'
```

What you should see:

```
{
  "total_properties": 18,
  "proved": 11,
  "falsified": 5,
  "timeout": 2,
  "protocols_verified": ["tls12", "tls13", "rtsp", "onvif", "mqtt"],
  "verification_time_seconds": 45.2,
  "critical_failures": 2
}
```

Step 2: List all verified TLS properties

```
curl -s "http://localhost:8100/v1/formal/results?scan_id=$SCAN_ID" \
  -H "Authorization: Bearer $TOKEN" \
  | jq '.data.results[]? | select((.protocol // "") | test("tls";
      "i")) | {device_ip, protocol, verdict}'
```

Step 3: Inspect one device's formal result

```
DEVICE_IP=$(curl -s "http://localhost:8100/v1/formal/$SCAN_ID" \
  -H "Authorization: Bearer $TOKEN" | jq -r
      '.data.results[0].device_ip')
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/$DEVICE_IP" \
  -H "Authorization: Bearer $TOKEN" | jq .
```

What you should see (for TLS 1.2 without client certificates):

```
{
  "status": "success",
  "data": {
    "property_name": "tls_mutual_auth",
    "property_type": "authentication",
    "protocol": "tls12",
    "result": "FALSIFIED",
    "description": "Both parties authenticate each other before
        session key exchange",
    "formal_statement": "forall s:session. authenticated(server,
        client, s) /\\ authenticated(client, server, s)",
    "falsification_method": "counterexample",
    "counterexample_id": "cex-001",
    "affected_devices": ["172.30.0.10", "172.30.0.11"],
    "severity": "medium",
    "rationale": "TLS 1.2 without client certificates only
        authenticates the server; the client is not authenticated to
        the server"
  }
}
```

Step 4: Examine the proof structure for a passing property

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/tls_server_auth"
        \
  -H "Authorization: Bearer $TOKEN" | jq '{
    result: .data.result,
    proof_method: .data.proof_method,
    proof_depth: .data.proof_depth,
    invariants_checked: .data.invariants_checked
  }'
```

Discussion Questions:

1. The `tls_mutual_auth` property is FALSIFIED because TLS 1.2 without client certificates does not authenticate the client to the server. Why is this considered a "medium" severity rather than "critical"? In which IoT deployment scenarios would the lack of client authentication be a critical security failure?

2. The formal statement `forall s:session. authenticated(server, client, s) /\ authenticated(client, server, s)` is a universally quantified property over all sessions. What does the `forall` quantifier mean for the verification process? Could the property be true for some sessions and false for others?

3. The verifier checked 18 properties in 45 seconds. ProVerif uses symbolic execution (Dolev-Yao adversary model). What assumptions does this model make that might not hold in practice, and what real-world attacks would fall outside the model?

4. Two properties timed out. What causes a formal verification timeout, and what should you do with a timeout result -- does it mean the property holds, or that you don't know?

> **Troubleshooting:** If `protocols_verified` is empty, the formal verifier requires Phase 3 protocol transcript data. Ensure a scan completed with protocol enrichment. If verification takes > 120 seconds, the system may be under load -- reduce scope with `?protocols=tls` query parameter.

Exercise 2: Check Authentication Property (Device to Server)

(See Slides 026-032 for authentication property formalization and ProVerif query syntax)

Step 1: Get server authentication result across all devices

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/server_authentication"
        \
  -H "Authorization: Bearer $TOKEN" | jq .
```

Step 2: Check per-device authentication status

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/server_authentication"
        \
  -H "Authorization: Bearer $TOKEN" | jq '.data.per_device_results[] |
        {ip: .device_ip, result: .result, protocol: .protocol}'
```

Step 3: Examine the ProVerif query that defines this property

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/server_authentication"
      \
  -H "Authorization: Bearer $TOKEN" | jq '.data.proverif_query'
```

What you should see:

```
{
  "query": "query x:key; event(serverAuthenticated(x)) ==>
       event(clientSentHello(x))",
  "interpretation": "If the server is authenticated with key x, then
       the client must have previously sent a hello with the same
       key. An attacker cannot trigger serverAuthenticated without
       the client initiating."
}
```

Step 4: Identify devices that fail server authentication

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/server_authentication"
      \
  -H "Authorization: Bearer $TOKEN" | jq '.data.per_device_results[] |
       select(.result == "FALSIFIED")'
```

Discussion Questions:

1. The ProVerif query uses "events" (`event(serverAuthenticated(x))`). What is the role of events in ProVerif's process calculus, and how do they differ from regular function calls in a programming language?

2. A device at 172.30.0.15 fails server authentication verification. Looking at the device list from Phase 1, what device type is this likely to be, and why might that device type commonly fail server authentication?

3. Server authentication failure in TLS means the device accepts connections from servers without verifying the server's certificate. What specific attacks does this enable that would not be possible with proper certificate verification?

4. The counterexample for server authentication failure will show an attack trace. Without seeing the trace: predict what the trace looks like in terms of message sequence (hint: what does a classic man-in-the-middle attack look like in TLS without cert verification?).

> **Troubleshooting:** If all devices show PROVED for server_authentication, the lab may be using TLS 1.3 devices that enforce strict certificate verification. Use `?protocol=rtsp` to check RTSP authentication, which is more likely to show failures.

Exercise 3: Verify Forward Secrecy

(See Slides 033-040 for forward secrecy definition, ephemeral key exchange, and session key independence)

Step 1: Check forward secrecy property

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/forward_secrecy"
      \
  -H "Authorization: Bearer $TOKEN" | jq .
```

What you should see:


```
{
  "status": "success",
  "data": {
    "property_name": "forward_secrecy",
    "formal_statement": "forall s1 s2:session. s1 != s2 ==>
        session_key(s1) != session_key(s2) /\\ not(derives(attacker,
        session_key(s1), long_term_key))",
    "result": "PARTIALLY_PROVED",
    "proved_for": ["tls13", "tls12_ecdhe"],
    "falsified_for": ["tls12_rsa", "rtsp_basic"],
    "explanation": "TLS 1.2 with RSA key exchange does not provide
        forward secrecy; the session key can be derived from the RSA
        private key and recorded traffic"
  }
}
```


Step 2: Identify devices using non-forward-secret key exchange

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/forward_secrecy"
        \
  -H "Authorization: Bearer $TOKEN" | jq '.data.devices_without_fs'
```

Step 3: Quantify HNDL exposure from non-FS sessions

```
# Cross-reference with Phase 7 HNDL data if available
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/forward_secrecy"
        \
  -H "Authorization: Bearer $TOKEN" | jq '{
    devices_without_fs: (.data.devices_without_fs | length),
    protocols_affected: .data.falsified_for,
    hndl_risk_amplified: .data.hndl_risk_amplified
  }'
```

Step 4: Confirm TLS 1.3 forward secrecy proof

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/forward_secrecy"
        \
  -H "Authorization: Bearer $TOKEN" | jq
        '.data.proof_details["tls13"]'
```

Discussion Questions:

1. Forward secrecy is formally stated as: if an attacker knows the long-term key (e.g., RSA private key), they still cannot derive the session key for any past session. How does ECDHE achieve this property while RSA key exchange does not?

2. The property is PARTIALLY_PROVED: proved for TLS 1.3 and TLS 1.2 with ECDHE, but falsified for TLS 1.2 with RSA. What does a partial proof mean for operational risk -- should you treat all TLS as insecure, or is partial proof meaningful?

3. RTSP Basic authentication is in the `falsified_for` list. RTSP/Basic doesn't use TLS at all -- why is forward secrecy relevant, and what specifically is the forward secrecy failure in basic HTTP/RTSP authentication?

4. "Forward secrecy" protects past sessions from future key compromise. "Post-compromise security" (an additional property in Signal protocol) also protects future sessions from past key compromise. Does TLS 1.3 provide post-compromise security? Why or why not?

> **Troubleshooting:** If `proved_for` and `falsified_for` are both empty, forward secrecy requires TLS handshake data from Phase 1 enrichment (TLS certificate inspection). Verify TLS data exists: `curl -s "http://localhost:8100/v1/scanning/smart-scan/$SCAN_ID" -H "Authorization: Bearer $TOKEN" | jq '.data.hosts[0].tls_info'`

Exercise 4: Test Replay Protection

(See Slides 041-048 for replay protection, nonce/timestamp schemes, and freshness properties)

Step 1: Check replay protection across all protocols

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/replay_protection"
       \
  -H "Authorization: Bearer $TOKEN" | jq '{
    result: .data.result,
    protocols_vulnerable: .data.protocols_vulnerable,
    protocols_protected: .data.protocols_protected
  }'
```

Step 2: Get the replay attack counterexample for RTSP

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/replay_protection"
       \
  -H "Authorization: Bearer $TOKEN" | jq '.data.counterexamples[] |
       select(.protocol == "rtsp")'
```

What you should see:

```
{
  "protocol": "rtsp",
  "attack_type": "replay",
  "steps": [
    {"step": 1, "actor": "attacker", "action": "record RTSP SETUP
        request from client C"},
    {"step": 2, "actor": "attacker", "action": "replay RTSP SETUP
        request to server S after session teardown"},
    {"step": 3, "actor": "server", "action": "accepts replayed SETUP,
        establishes unauthorized stream"},
    {"step": 4, "actor": "attacker", "action": "receives camera stream
        without authentication"}
  ],
  "severity": "HIGH",
  "cvss_equivalent": 8.1
}
```

Step 3: Confirm MQTT QoS nonce handling

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/replay_protection"
       \
  -H "Authorization: Bearer $TOKEN" | jq '.data.protocols_protected'
```

Step 4: Quantify affected devices

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/replay_protection"
       \
  -H "Authorization: Bearer $TOKEN" | jq '.data.protocols_vulnerable[]
       as $p | {protocol: $p, device_count:
       (.data.affected_devices[$p] | length)}'
```

Discussion Questions:

1. Replay protection requires freshness: each message must be provably new (not a replay of a previous message). Name three different mechanisms for ensuring freshness and explain the tradeoffs of each in an IoT environment with limited memory and no reliable clock.

2. The RTSP replay attack establishes an unauthorized camera stream. In what scenarios is unauthorized camera access a safety issue (not just a privacy issue)?

3. TLS provides replay protection via the sequence number counter. If a TLS connection's sequence number wraps around (overflows), what happens to replay protection? When would this practically occur?

4. The formal model assumes the attacker can record and replay any message. In practice, what infrastructure does an attacker need to execute the RTSP replay attack, and how could network topology design mitigate this even without fixing the protocol?

> **Troubleshooting:** If RTSP replay shows as PROVED (not falsified), the lab cameras may be running RTSP over TLS (RTSPs), which adds replay protection. Check: `curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties" -H "Authorization: Bearer $TOKEN" | jq '.data.properties[] | select(.name | contains("rtsp"))'`

Exercise 5: Run Downgrade Attack Detection

(See Slides 049-056 for protocol version downgrade, POODLE/FREAK/Logjam, and version negotiation attacks)

Step 1: Check for downgrade attack vulnerabilities

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/downgrade_protection"
       \
  -H "Authorization: Bearer $TOKEN" | jq '.data'
```

What you should see:

```
{
  "property_name": "downgrade_protection",
  "result": "FALSIFIED",
  "attacks_detected": [
    {
      "attack": "POODLE",
      "description": "SSLv3 fallback enables CBC padding oracle",
      "affected_devices": ["172.30.0.10", "172.30.0.14"],
      "severity": "HIGH"
```

```
    },
    {
      "attack": "FREAK",
      "description": "Export-grade RSA key exchange (512-bit)
        accepted",
      "affected_devices": ["172.30.0.30"],
      "severity": "HIGH"
    }
  ],
  "downgrade_resistant_devices": ["172.30.0.31", "172.30.0.32"]
}
```

Step 2: Get the POODLE counterexample trace

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/counterexamples" \
  -H "Authorization: Bearer $TOKEN" | jq '.data.counterexamples[] |
        select(.attack_name == "POODLE")'
```

Step 3: Check for TLS Renegotiation vulnerability

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/tls_renegotiation_safe
        \
  -H "Authorization: Bearer $TOKEN" | jq '{result: .data.result,
        affected_devices: .data.affected_devices}'
```

Step 4: Verify TLS_FALLBACK_SCSV support

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/fallback_scsv"
        \
  -H "Authorization: Bearer $TOKEN" | jq '{
    devices_supporting: (.data.supporting_devices | length),
    devices_not_supporting: (.data.non_supporting_devices | length),
    result: .data.result
  }'
```

Discussion Questions:

1. POODLE (Padding Oracle On Downgraded Legacy Encryption) requires a CBC-mode cipher with predictable padding. TLS 1.3 only uses AEAD ciphers (GCM, ChaCha20-Poly1305). Why does switching to TLS 1.3 completely eliminate POODLE, while patching TLS 1.2 against POODLE is more complex?

2. FREAK (Factoring RSA Export Keys) exploits the existence of "export-grade" 512-bit RSA keys, which were mandated by US export control regulations in the 1990s. Why did these weak keys survive in implementations for 20+ years after the export restrictions were lifted?

3. The TLS_FALLBACK_SCSV (Signaling Cipher Suite Value) prevents downgrade attacks by signaling to the server that the client is sending a non-preferred version. If both client and server support TLS 1.3, but the client sends TLS 1.2 as a fallback (perhaps due to a network device that strips TLS 1.3 extensions), what should the server do upon seeing TLS_FALLBACK_SCSV?

4. You identified that 172.30.0.10 supports SSLv3 fallback (POODLE-vulnerable). Propose two remediation options: one that requires firmware update and one that does not. Which would you recommend for an immediate (< 24 hour) fix?

> **Troubleshooting:** If no downgrade attacks are detected (all PROVED), the lab cameras may have already been hardened. Check their TLS minimum version: `curl -s "http://localhost:8100/v1/formal/$SCAN_ID/models/172.30.0.10" -H "Authorization: Bearer $TOKEN" | jq '.data.tls_version_support'`

Exercise 6: Check RFC Conformance Score

(See Slides 057-064 for RFC grammar rules, conformance testing, and coverage metrics)

Step 1: Get RFC conformance scores for all protocols

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/rfc" \
  -H "Authorization: Bearer $TOKEN" | jq '.data.protocols[] |
        {protocol: .protocol, score: .conformance_score, rules_passed:
        .rules_passed, rules_failed: .rules_failed}'
```

Step 2: Examine specific failing RFC rules

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/rfc" \
  -H "Authorization: Bearer $TOKEN" | jq '.data.protocols[] |
        select(.conformance_score < 0.8) | .failing_rules'
```

What you should see:

```
[
  {
    "rule_id": "RFC7230-3.1.1",
    "description": "Request-Line must not contain bare CR (carriage
        return) without LF",
    "observed": "GET /stream\\r HTTP/1.0 (bare CR before path
        terminator)",
    "expected": "GET /stream HTTP/1.0\\r\\n",
    "severity": "MEDIUM",
    "attack_relevance": "HTTP request smuggling via CR injection"
  },
  ...
]
```

Step 3: Get grammar rule coverage statistics

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/rfc" \
  -H "Authorization: Bearer $TOKEN" | jq '{
    total_rules_checked: .data.total_rules_checked,
    coverage_percent: .data.rule_coverage_percent,
    uncovered_rules: (.data.uncovered_rules | length)
  }'
```

Step 4: Find the lowest-conformance device

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/rfc" \
  -H "Authorization: Bearer $TOKEN" | jq '.data.protocols |
        sort_by(.conformance_score) | .[0]'
```

Discussion Questions:

1. RFC conformance testing checks whether a device implements the protocol according to the specification. A device can be "RFC-conformant" but still vulnerable, and a device can be "non-conformant" but still secure. Give an example of each scenario.

2. The rule RFC7230-3.1.1 (no bare CR in HTTP requests) has attack relevance "HTTP request smuggling." Explain how a bare CR in an HTTP request could lead to request smuggling against a downstream proxy or web application firewall.

3. Grammar rule coverage is 73% -- only 73% of the RFC rules are exercised by the observed protocol traffic. What are the security implications of the uncovered 27% of rules, and how would you increase coverage?

4. The Phase 3 grammar inference engine learned a grammar from observed protocol traffic. The Phase 11 RFC conformance engine compares observed behavior against the RFC-specified grammar. What is the relationship between the inferred grammar (Phase 3) and the RFC grammar (Phase 11), and under what conditions would they diverge?

> **Troubleshooting:** If conformance scores all show 1.0 (100%), the RFC checker may not have sufficient protocol transcript data. Ensure Phase 3 protocol scanning was enabled: `BREAKWATER_PROTOCOL_SECURITY_ENABLED=true`.

Exercise 7: View Attack Traces (Counterexamples)

(See Slides 065-072 for counterexample generation, attack trace interpretation, and remediation guidance)

Step 1: List all generated counterexamples

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/counterexamples" \
  -H "Authorization: Bearer $TOKEN" | jq '.data.counterexamples[] |
        {id: .counterexample_id, property: .property_violated, attack:
        .attack_type, severity: .severity}'
```

Step 2: Get the full attack trace for the most severe counterexample

```
# Get the highest-severity counterexample
CEX_ID=$(curl -s
        "http://localhost:8100/v1/formal/$SCAN_ID/counterexamples" \
  -H "Authorization: Bearer $TOKEN" | jq -r '.data.counterexamples |
        sort_by(.severity_score) | reverse | .[0].counterexample_id')

curl -s "http://localhost:8100/v1/formal/$SCAN_ID/counterexamples" \
  -H "Authorization: Bearer $TOKEN" | jq ".data.counterexamples[] |
        select(.counterexample_id == \"$CEX_ID\")"
```

Step 3: Parse the attack trace steps

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/counterexamples" \
  -H "Authorization: Bearer $TOKEN" | jq ".data.counterexamples[] |
        select(.counterexample_id == \"$CEX_ID\") | .trace.steps[] |
        {step: .step_number, actor: .actor, message:
        .message_description}"
```

Step 4: Map counterexample to MITRE technique

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/counterexamples" \
  -H "Authorization: Bearer $TOKEN" | jq ".data.counterexamples[] |
        select(.counterexample_id == \"$CEX_ID\") | .mitre_mapping"
```

Discussion Questions:

1. A counterexample is a specific execution trace that violates a security property. How does a counterexample differ from a proof? What does the existence of a counterexample tell you that a failed penetration test (no vulnerability found) does not?

2. The attack trace shows the attacker as an explicit actor ("attacker intercepts message M"). In a real network, who or what plays the "attacker" role, and what network position does the attacker need to be in to execute the attack steps?

3. ProVerif generates counterexamples symbolically -- the "attacker" is an abstract entity with the power to intercept, replay, and construct any message. A real attacker is constrained by their actual network position, tool limitations, and detection risk. How do you translate a symbolic counterexample into a practical attack assessment?

4. You have 5 counterexamples and only resources to fix 2 this sprint. How do you prioritize -- by property violated, by severity, by number of affected devices, or by ease of remediation? Propose a multi-factor prioritization formula.

> **Troubleshooting:** If `total_counterexamples: 0`, either all properties were proved (unlikely for a real network) or verification did not complete. Check: `curl -s "http://localhost:8100/v1/formal/$SCAN_ID/verify" -H "Authorization: Bearer $TOKEN" | jq '.data.summary.falsified'`

Exercise 8: Verify Matter SPAKE2+ Properties

(See Slides 073-080 for Matter protocol, SPAKE2+ password-authenticated key exchange, and commissioning security)

Step 1: Check SPAKE2+ password authentication property

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/spake2plus_auth"
        \
  -H "Authorization: Bearer $TOKEN" | jq '.data'
```

Step 2: Verify the Key Confirmation property

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/spake2plus_key_confirm
        \
  -H "Authorization: Bearer $TOKEN" | jq '{result: .data.result,
       explanation: .data.explanation}'
```

Step 3: Check dictionary attack resistance

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/spake2plus_dict_resist
        \
  -H "Authorization: Bearer $TOKEN" | jq .
```

Step 4: List Matter-capable devices in the scan

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/models" \
  -H "Authorization: Bearer $TOKEN" | jq '.data.models[] |
        select(.protocol_family == "matter") | {ip: .device_ip,
        vendor: .device_vendor}'
```

Discussion Questions:

1. SPAKE2+ is a Password-Authenticated Key Exchange (PAKE) protocol. How does PAKE differ from a standard key exchange (like ECDH) that is not password-authenticated? What attack does PAKE prevent that ECDH does not?

2. The Matter commissioning process uses a 20-digit setup code (embedded in a QR code on the device). SPAKE2+ derives the session key from this setup code. What is the entropy of a 20-digit numeric setup code, and how does this compare to the entropy of a typical Wi-Fi password?

3. "Dictionary attack resistance" for SPAKE2+ means an eavesdropper who captures the SPAKE2+ handshake cannot determine the password without performing an online brute-force attack (requiring interaction with the device for each guess). Why is online brute force significantly harder to execute than offline brute force from captured handshake data?

4. If a Matter device is manufactured with a hardcoded setup code that is the same for all units of that model (a common cost-cutting measure), what does this do to SPAKE2+'s security guarantees? What MITRE technique does this enable?

> **Troubleshooting:** If `spake2plus_auth` returns "not_applicable" (no Matter devices found), use the ONVIF WS-Security property instead: `curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/onvif_ws_security" -H "Authorization: Bearer $TOKEN" | jq .`

Exercise 9: Run EDHOC Draft Detection

(See Slides 081-086 for EDHOC (Ephemeral Diffie-Hellman Over COSE) and constrained IoT security)

Step 1: Check for EDHOC protocol support

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/edhoc_support"
        \
  -H "Authorization: Bearer $TOKEN" | jq .
```

Step 2: Check for CoAP/OSCORE formal properties

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/coap_oscore_auth"
        \
  -H "Authorization: Bearer $TOKEN" | jq '{result: .data.result,
        devices_using_coap: .data.affected_devices}'
```

Step 3: Verify COSE (CBOR Object Signing and Encryption) integrity

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID/properties/cose_integrity"
        \
  -H "Authorization: Bearer $TOKEN" | jq '.data'
```

Step 4: Compare EDHOC security properties to TLS 1.3

```
curl -s "http://localhost:8100/v1/formal/results?scan_id=$SCAN_ID" \
  -H "Authorization: Bearer $TOKEN" \
  | jq '[.data.results[]? | select((.protocol // "") |
        test("edhoc|tls13"; "i")) | {device_ip, protocol, verdict}]'
```

Discussion Questions:

1. EDHOC (Ephemeral Diffie-Hellman Over COSE) is designed for constrained IoT devices. What constraints make standard TLS 1.3 impractical for Class 0/1 constrained devices (< 10 KB RAM, < 100 KB flash), and how does EDHOC address each constraint?

2. OSCORE (Object Security for Constrained RESTful Environments) provides end-to-end application-layer security for CoAP, independently of transport security. Why would you need application-layer security (OSCORE) in addition to transport-layer security (DTLS)?

3. COSE (CBOR Object Signing and Encryption) is the JSON Web Signature/Encryption (JWS/JWE) equivalent for constrained devices. What binary encoding advantages does CBOR have over JSON for IoT applications with tight bandwidth constraints?

4. Most IoT devices in the lab use TLS-based protocols, not EDHOC/CoAP. Given that EDHOC is a newer, more efficient protocol, what barriers prevent widespread adoption of EDHOC in commercial IoT devices?

> **Troubleshooting:** EDHOC support requires CoAP-enabled devices in the scan. In the IoT sim, MQTT and HTTP devices are common; CoAP devices are rare. If EDHOC shows "no devices found," proceed with the MQTT formal properties instead.

Exercise 10: Export Formal Verification Report

(See Slides 087-098 for verification report structure, audit evidence, and remediation integration)

Step 1: Generate the formal verification summary

```
curl -s "http://localhost:8100/v1/formal/$SCAN_ID" \
  -H "Authorization: Bearer $TOKEN" | jq '.data'
```

What you should see:

```
{
  "scan_id": "a3f1c2d4-...",
  "report_generated_at": "2026-03-05T16:00:00Z",
  "total_devices_verified": 18,
  "total_properties_checked": 18,
  "critical_findings": 2,
  "high_findings": 3,
  "medium_findings": 4,
  "low_findings": 2,
  "overall_security_posture": "REQUIRES_IMMEDIATE_ATTENTION",
  "top_priority_finding": "RTSP replay protection absent on 8
        cameras",
```

```
  "estimated_remediation_effort_hours": 42
}
```

Step 2: Get attack traces

```
curl -s "http://localhost:8100/v1/formal/traces?scan_id=$SCAN_ID" \
  -H "Authorization: Bearer $TOKEN" | jq '.data.traces[:5]'
```

Step 3: Get RFC conformance evidence

```
curl -s "http://localhost:8100/v1/formal/conformance?scan_id=$SCAN_ID"
       \
  -H "Authorization: Bearer $TOKEN" | jq '.data'
```

Step 4: Export device-level results for integration

```
curl -s "http://localhost:8100/v1/formal/results?scan_id=$SCAN_ID" \
  -H "Authorization: Bearer $TOKEN" \
  | jq '{scan_id: .data.scan_id, devices: (.data.results | length),
        red: ([.data.results[]? | select(.verdict == "RED")] |
        length)}'
```

Discussion Questions:

1. The formal verification report claims 18 devices were "formally verified." What does this mean precisely? Is it a guarantee that these devices are secure, or something more limited?

2. The report estimates "42 hours remediation effort." What assumptions go into this estimate, and under what circumstances would the actual effort be significantly higher?

3. How would you integrate formal verification findings with a SOC's SOAR (Security Orchestration, Automation, and Response) platform? What automated responses could a SOAR trigger based on formal verification results?

4. Formal verification catches "all possible" attacks within the model's assumptions. Phase 3's fuzzing catches "many real" attacks by sending unexpected inputs. A Phase 5 penetration test catches "confirmed exploitable" vulnerabilities by actually executing attacks. What does each approach contribute to a comprehensive security assessment, and what does each approach miss?

> **Troubleshooting:** If the report shows `total_devices_verified: 0`, formal verification has not been run for this scan. Run Exercise 1 first, then re-request the report. If `format=jira` returns 400, Jira integration requires `JIRA_URL` and `JIRA_TOKEN` environment variables -- use `format=json` instead.

Lab Wrap-Up

Across these 10 exercises, you have applied formal verification to the IoT lab network:

1. **Verified** TLS properties using symbolic model checking with a Dolev-Yao adversary
2. **Checked** authentication properties across all devices with per-device granularity
3. **Proved** (and falsified) forward secrecy for different key exchange algorithms
4. **Detected** replay protection failures with concrete attack traces
5. **Identified** downgrade vulnerabilities including POODLE and FREAK
6. **Scored** RFC conformance with rule-level coverage analysis

7. **Analyzed** counterexample attack traces and mapped them to MITRE techniques
8. **Verified** modern IoT protocol properties (SPAKE2+, EDHOC, OSCORE)
9. **Compared** formal security properties across protocol families
10. **Exported** a comprehensive verification report with remediation guidance

The critical insight of Phase 11: formal verification does not replace testing -- it complements it. Testing catches bugs in specific implementations. Formal verification proves (or disproves) properties of the protocol design itself. A formally verified protocol can still have a buggy implementation. A formally unverified protocol can have a correct implementation. Both layers of analysis are necessary for high-assurance security.

Clean-up:

```
# Verification results are stored with the scan -- no cleanup required
echo "Formal verification results are preserved with scan $SCAN_ID"
```