Pipeline-Oriented Security Analysis: Advanced Techniques

Model checking, symbolic execution, and attack graph generation for pipeline programs

Lateralus Language

bad-antics · April 2026 · nullsec / Lateralus Language Research

ABSTRACT This paper extends the pipeline security analysis framework with three advanced techniques: model checking pipeline state machines against LTL security properties, symbolic execution of pipeline stages to discover edge-case vulnerabilities, and automated attack graph generation from pipeline topology. These techniques are complementary to the taint tracking and trust boundary auditing described in the companion paper and are intended for high-assurance applications where manual review alone is insufficient.

1. Model Checking Pipeline State Machines

A pipeline that processes requests with authentication, authorization, and auditing can be modeled as a finite state machine. Model checking verifies that the state machine satisfies temporal logic properties, such as 'authorization is always checked before resource access'.

```
// Pipeline state machine model
enum RequestState {
    Received,
    Authenticated,
    Authorized,
    ResourceAccessed,
    Audited,
}

// LTL property: authorization always precedes access
// G(ResourceAccessed -> P(Authorized))
// 'Globally, if resource is accessed, authorized must have happened first'
```

The Lateralus security analyzer tool translates pipeline programs to state machine models and checks the specified LTL properties using an on-the-fly model checker. Property violations are reported as counterexample traces through the pipeline.

2. Symbolic Execution of Pipeline Stages

Symbolic execution runs a function with symbolic inputs (variables representing all possible values) and tracks path conditions. When a path leads to a security-relevant state (a panic, an unchecked unwrap, or a taint violation), the solver finds a concrete input that exercises that path.

```
// Symbolic execution of a pipeline stage
ltl symex nullsec::parse::http_headers
# Analyzes all paths through http_headers
# Report:
#   FOUND: path leads to panic
#   Condition: header_line.len() == 0
#   Concrete input: b"\r\n"
#   Fix: add guard: if line.is_empty() { return Err(ParseError::EmptyLine) }
```

Symbolic execution finds path-specific bugs that fuzzing misses because random generation may not reach the specific combination of values that triggers the bug.

3. Attack Graph Generation

An attack graph models the paths an adversary can take through a system, starting from an initial compromise and reaching a target state (data exfiltration, privilege escalation, etc.). Pipeline topology provides the structure for the attack graph.

```
// Generate attack graph for a web application pipeline
ltl security attack-graph src/web_app.lt
# Analyzes pipeline structure
# Outputs: attack_graph.dot (Graphviz)
#   Node: user_input (untrusted)
#   Edge: parse_json -> validate -> db_query
#   Attack: skip validate -> direct db_query = SQL injection
#   Mitigation: add Tainted<T> label to user_input
```

The attack graph generator identifies which pipeline stages are reachable without passing through a required security check. These are the attack paths; mitigations are expressed as additional required stages in the pipeline.

4. Property-Based Security Testing

Property-based testing verifies that a stage satisfies a security property for all inputs, not just known test cases. nullsec provides a property library for common security properties:

```
// Property: sanitizer is idempotent (safe(safe(x)) == safe(x))
#[property]
fn html_escape_idempotent(s: str) -> bool {
    let once   = html_escape(s.clone());
    let twice  = html_escape(once.clone());
    once == twice
}

// Property: parser + printer roundtrip
#[property]
fn json_roundtrip(v: JsonValue) -> bool {
    parse_json(print_json(v.clone())) == Ok(v)
}
```

Property failures are shrunk automatically: the framework finds the smallest input that violates the property, making debugging the failure much easier.

5. Compositional Security Verification

When a pipeline is composed of verified stages, can the composed pipeline be considered secure? Compositional security verification answers this question using the security type system.

If each stage has a verified security specification (a pre/post condition expressed in the type system), then the composed pipeline's specification is the composition of the individual specifications. This is the security analog of the pipeline algebraic laws: just as stage fusion preserves behavior, specification composition preserves security properties.

```
-- Compositional security: if each stage is secure, the pipeline is
secure(stage_1) ∧ secure(stage_2) ∧ ... ∧ secure(stage_n)
■ secure(pipe { |> stage_1 |> stage_2 ... |> stage_n })
```

6. Threat Modeling from Pipeline Topology

STRIDE threat modeling (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) can be applied to pipeline topology automatically. The nullsec threat modeler analyzes the pipeline graph and generates a STRIDE analysis for each stage boundary:

```
ltl security stride src/payment_pipeline.lt
# Stage boundary: user_input -> validate_card
```

```
#   S (Spoofing):    Input claims card ownership — check: identity assertion present
#   T (Tampering):   Card number modified in transit — check: HTTPS enforced
#   R (Repudiation): No audit log of card validation — ISSUE: add audit stage
#   I (Disclosure):  Card number logged in error messages — ISSUE: mask PAN
#   D (DoS):         No rate limiting — ISSUE: add rate_limiter stage
#   E (EoP):         No privilege check — OK: user context verified
```

7. Integration with CI/CD

The security analysis tools integrate with standard CI/CD pipelines. The nullsec CI action runs taint analysis, trust boundary audit, and property tests on every pull request:

```
# .github/workflows/security.yml
- name: Taint analysis
  run: ltl security taint src/
- name: Trust boundary audit
  run: ltl security audit --trust-boundaries src/
- name: Property tests
  run: ltl test --properties src/
- name: Fuzz (10 minutes)
  run: ltl fuzz --duration 600 src/
```

The CI action blocks merges when any security analysis reports a finding with severity HIGH or CRITICAL. MEDIUM findings generate warnings but do not block merges.

8. Case Study: Auditing a Login Pipeline

We applied the full analysis suite to the nullsec login pipeline (200 lines, 6 stages). The analysis found:

- **Taint analysis**: the username was used as a SQL query parameter without the Tainted label, allowing injection if the downstream stage changed. Fixed by adding the label.
- **Symbolic execution**: a panic reachable when the password hash was exactly 0 bytes (an impossible input in practice but technically reachable). Fixed with a bounds check.
- **STRIDE analysis**: no audit log for failed login attempts (repudiation). Fixed by adding an audit stage.

The analysis took 4 minutes and found three issues that manual review had missed. All three were fixed before the code was deployed.