Pipeline-Oriented Security Analysis

Using Lateralus pipelines to model, audit, and test security properties

Lateralus Language

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

ABSTRACT Security analysis of software systems involves tracing data flows through components, identifying trust boundaries, and verifying that sensitive data does not escape its authorized scope. The Lateralus pipeline model makes these analyses more tractable: data flows are explicit pipeline stages, trust boundaries correspond to pipeline operator changes, and the type system can encode data sensitivity labels. This paper describes three pipeline-oriented security analysis techniques: taint tracking via type labels, trust boundary auditing via pipeline shape inspection, and automated fuzzing of pipeline stages.

1. Pipelines as Security Models

A Lateralus pipeline models a security-relevant computation as a sequence of typed transformations. Each stage corresponds to a component in the system; the stage's input and output types correspond to the component's expected interface.

This makes security properties easy to state: 'sensitive data must be sanitized before leaving the trust boundary' becomes 'a pipeline stage labeled Sanitizer must appear before any stage labeled ExternalOutput'. The compiler can check this statically.

2. Taint Tracking with Type Labels

Type labels mark values as tainted (from untrusted sources) or clean (sanitized). The type system propagates labels through pipeline stages:

```
// Type labels for taint tracking
sealed enum Tainted<T> { T }   // wraps a value of type T
sealed enum Clean<T>   { T }   // sanitized value of type T

// Sanitizer stage: converts Tainted to Clean
fn html_escape(s: Tainted<str>) -> Clean<str>

// External output: accepts only Clean values
fn send_to_browser(s: Clean<str>) -> Result<(), IoError>

// Pipeline: taint tracking enforced by types
let input: Tainted<str> = user_input();
let safe = input
    |>  html_escape      // Tainted<str> -> Clean<str>
    |?> send_to_browser  // Clean<str> -> Result<(), _>
```

Attempting to pass a Tainted<str> directly to send_to_browser is a compile error: the types do not match. The sanitizer stage is enforced by the type system, not by runtime checks or code review.

3. Trust Boundary Auditing

A trust boundary is a point where data moves between two components with different trust levels. In a Lateralus pipeline, trust boundaries can be marked with an annotation that the auditing tool checks:

```
// Trust boundary annotation
#[trust_boundary(from = "user", to = "kernel")]
fn handle_syscall(req: UserRequest) -> KernelResponse { ... }

// Audit tool command
ltl security audit --trust-boundaries src/
```

```
# Finds all #[trust_boundary] annotations and checks:
# 1. Both sides have non-empty type signatures
# 2. Input types are sanitized before the boundary
# 3. Error propagation is explicit (|?> not unchecked unwrap)
```

The audit tool generates a trust boundary map — a graph where nodes are trust domains and edges are pipeline stages that cross between them. Security reviewers use the map to focus their manual review on the highest-risk transitions.

4. Automated Pipeline Fuzzing

Pipeline stages have typed inputs and outputs, making them ideal targets for automated fuzzing. The nullsec fuzzer generates values of the stage's input type using type-directed random generation and runs the stage, checking for panics, type invariant violations, or unexpected errors.

```
// Fuzzing a pipeline stage
ltl fuzz nullsec::parse::http_request
# Generates random Vec<u8> inputs (the stage's input type)
# Runs 1,000,000 iterations
# Reports crashes and unexpected Err variants
```

Type-directed generation is more efficient than byte-level fuzzing because it produces structurally valid inputs: a stage that accepts a JsonObject receives valid JSON, not random bytes that would be rejected by the parser. This focuses fuzzing on the stage's actual logic.

4.1 Differential Fuzzing

For stages with reference implementations, differential fuzzing compares the Lateralus stage output against the reference for the same input. Discrepancies indicate bugs in either implementation.

5. Information Flow Analysis

The type label system can be extended to full information flow control: labels carry security lattice values (Public < Private < Secret < TopSecret), and the type rules enforce that information never flows from a higher security level to a lower one.

```
enum SecurityLevel { Public, Private, Secret, TopSecret }

// A labeled value
struct Labeled<T, const LEVEL: SecurityLevel> { value: T }

// Declassification requires explicit authorization
fn declassify<T>(v: Labeled<T, Secret>, auth: DeclassifyCap)
    -> Labeled<T, Public>

// Compilation error: leaking Secret to Public without declassify
let secret: Labeled<str, Secret> = db::fetch_secret();
let _: Labeled<str, Public> = secret;  // error: Secret > Public
```

6. Static SQL Injection Detection

SQL injection is detectable at compile time when query construction is expressed as a pipeline. A query builder stage that accepts tainted string values is a compile error:

```
// Safe: parameterized query (accepts any str, escapes it)
fn query_by_name(name: str) -> Vec<Row> {
    db::execute("SELECT * FROM users WHERE name = ?", [name])
```

```
}

// Unsafe: string interpolation (detected by the Tainted type)
fn query_unsafe(name: Tainted<str>) -> Vec<Row> {
    let sql = format!("SELECT * FROM users WHERE name = '{}'", name);
    //                                                    ^^^^
    // error: cannot use Tainted<str> in SQL string interpolation
    db::execute_raw(sql)
}
```

7. Audit Trail Integration

Every pipeline in a security-critical application can be wrapped with an audit transformer that logs all inputs, outputs, and errors to an append-only, signed audit trail:

```
fn with_audit(label: str, p: Pipeline<A, B>) -> Pipeline<A, B> {
    pipe {
        |> audit::log_input(label)
        |> p
        |> audit::log_output(label)
    }
}

let audited_login = login_pipeline
    |> with_audit("user_login")
// Every login attempt is logged with input hash and outcome
```

The audit log uses the same hash-chained ledger mechanism as the element-115-drive telemetry system (paper 14). Each entry is signed with the node's Ed25519 key, making tampering detectable.

8. Limitations and Future Work

Current limitations of the pipeline security analysis approach:

- The taint label system requires explicit typing of all sanitizer stages. Implicit sanitization (e.g., integer parsing that inherently prevents injection) is not automatically credited.
- The trust boundary auditor does not verify that the source code matches the binary — only that the annotations are consistent with the code as written.
- The fuzzer does not yet model multi-stage pipelines: it fuzzes individual stages, not sequences of stages that might interact.

Future work: extending the taint system to cover more of the standard library automatically (e.g., all integer-parsing functions produce Clean output by default), integrating the audit trail with SIEM systems via the OCSF schema, and adding a property-based test mode to the fuzzer.