Pipeline-Native Penetration Testing

Automating reconnaissance, exploitation, and reporting with Lateralus pipelines

Lateralus Language

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

ABSTRACT Penetration testing involves a sequence of data-driven phases: reconnaissance, scanning, enumeration, exploitation, and reporting. Each phase transforms data from the previous phase. This is a natural fit for the Lateralus pipeline model. This paper describes how the nullsec toolkit implements the penetration testing lifecycle as a typed pipeline, with emphasis on the advantages over traditional Bash/Python tooling: compile-time composability, structured output for automated report generation, and reproducible test runs via sealed pipeline definitions.

1. The Penetration Testing Pipeline

A penetration test has five phases that correspond naturally to pipeline stages:

```
let findings = engagement_scope
    |>  recon::passive_discovery       // OSINT, certificates, DNS
    |>> scan::port_and_service          // active scanning
    |>  enum::enumerate_attack_surface // web paths, APIs, services
    |?> exploit::test_vulnerabilities   // attempt exploitation
    |>  report::generate_findings       // structured report
```

Each stage produces typed output that the next stage consumes without text parsing. The entire engagement is a single pipeline expression that can be sealed (frozen) for reproducible re-execution and audited for scope creep.

2. Passive Reconnaissance

Passive reconnaissance collects information about the target without generating network traffic to the target itself. nullsec's recon::passive_discovery stage queries public data sources:

```
fn passive_discovery(scope: EngagementScope) -> ReconResult {
    scope
        |>  cert_transparency::search_domains   // crt.sh
        |>  dns::enumerate_subdomains           // bruteforce + zone transfer
        |>  osint::search_employees             // LinkedIn public profiles
        |>  shodan::query_exposed_services      // Shodan API
        |>  wayback::find_old_paths             // Wayback Machine
        |>  deduplicate_and_merge
}
```

The result is a typed ReconResult containing all discovered subdomains, IP ranges, technology fingerprints, and employee names — structured data ready for the scanning stage.

3. Active Scanning

Active scanning generates traffic to the target. The scanning stage takes the recon result and expands it with service information:

```
fn port_and_service(recon: ReconResult) -> Vec<HostProfile> {
    recon.ip_ranges
        |>  scan::syn_scan({ ports: COMMON_PORTS, rate: 1000 })
        |>  filter(|h| h.has_open_ports())
        |>  service::detect_versions
        |>  web::fingerprint_if_http
        |>  collect()
}
```

The syn_scan stage is async (|>> in the top-level pipeline) because network scanning is I/O-bound and benefits from concurrent execution. The async model allows scanning multiple hosts simultaneously without explicit threading code.

4. Vulnerability Testing

The exploitation stage tests the discovered services for known vulnerabilities. This is the most sensitive stage; it is wrapped with scope checking to prevent out-of-scope exploitation:

```
fn test_vulnerabilities(hosts: Vec<HostProfile>) -> Result<Vec<Finding>, ScopeError> {
    hosts
        |>  scope::filter_in_scope
        |>  vuln::check_cve_database
        |?> vuln::test_authentication_weaknesses
        |?> vuln::test_injection_points
        |>  scope::verify_all_in_scope   // double-check before return
        |>  collect()
}
```

The scope checking stages prevent the pipeline from testing hosts outside the engagement scope. If any host fails the scope check, the |?> operator propagates a ScopeError that aborts the pipeline and logs the incident to the audit trail.

5. Automated Report Generation

The report stage transforms typed findings into a structured engagement report. Because findings are typed (not text), the report generator can aggregate, deduplicate, and prioritize them automatically:

```
fn generate_findings(findings: Vec<Finding>) -> EngagementReport {
    findings
        |>  deduplicate_by_cve
        |>  sort({ by: .severity, descending: true })
        |>  group_by_host
        |>  report::build_executive_summary
        |>  report::build_technical_details
        |>  report::export_pdf
}
```

The report is generated from the typed finding data; no manual editing is required for the technical section. The executive summary is generated from aggregated severity counts and affected business functions.

6. Scope Enforcement

Scope enforcement is the critical safety mechanism in automated penetration testing. nullsec implements scope as a type-level constraint: only InScope<T> values can be passed to exploitation functions.

```
// Scope-checked wrapper type
sealed enum InScope<T> { T }

// The only way to create an InScope value is through scope::check
fn scope_check(host: HostProfile, scope: &EngagementScope)
    -> Result<InScope<HostProfile>, ScopeError>

// Exploitation functions only accept InScope values
fn test_sql_injection(host: InScope<HostProfile>) -> Vec<SqliFinding>
```

An exploitation function that accidentally receives an out-of-scope host is a compile error, not a runtime error. This provides a much stronger guarantee than a runtime scope check that could be accidentally bypassed.

7. Engagement Archiving and Reproducibility

A sealed engagement pipeline can be archived and re-executed to reproduce findings. The archive contains the pipeline definition, the engagement scope, the tool versions, and the execution timestamps:

```
// Archive an engagement for reproduction
ltl pentest archive engagement_2026_04.lt
# Creates: engagement_2026_04.archive
#   contents: pipeline definition (frozen)
#             scope definition
#             tool versions (SHA-256 hashes)
#             timestamp of original execution
```

Reproduced engagements run the same pipeline with the same tool versions against the same scope. If findings differ, the difference is attributed to changes in the target environment, not tool variation.

8. Legal and Ethical Considerations

Pipeline-native penetration testing automates steps that, when performed without authorization, constitute computer fraud. The nullsec toolkit includes mandatory scope enforcement and authorization verification that cannot be disabled without modifying the source code.

Users of the nullsec toolkit are responsible for ensuring they have written authorization for all engagement targets. The toolkit does not provide and explicitly discourages any capability for unauthorized access, persistence, or data exfiltration. The exploitation stage is limited to detection (confirming a vulnerability is present) and does not include active exploit payloads.