Systematic Smart Contract Auditing: Methods and Tools

Zach Kelling

Lux Industries

2025

Abstract

A smart contract audit is a systematic search for vulnerabilities in code that controls money. This paper presents a complete audit methodology: static analysis with Slither, fuzz testing with Foundry, pattern detection with Aderyn, symbolic execution with Halmos, and manual review. We describe the methodology applied to Lux Network's 50 contract domains (AMM, bridge, vault, staking, governance, oracle, token, NFT, lending, and 41 more), comprising 832 tests and 122 findings across five severity levels. For each vulnerability class—reentrancy, flash loan attacks, oracle manipulation, access control bypass, integer overflow, frontrunning, and others—we provide the detection method, a Solidity example, the fix, and the tool that catches it. This is a practitioner's manual: every technique described here is used in production on every Lux Network smart contract deployment.

Contents

1 The Audit Process

1.1 Why Audit

Smart contracts are immutable programs that custody funds. A bug in a smart contract cannot be patched after deployment (without proxy patterns, which introduce their own risks). The asymmetry is extreme: the attacker needs to find one bug; the defender must eliminate all bugs.

1.2 Five-Phase Methodology

Our audit process has five sequential phases. Each phase catches different bug classes. Skipping any phase leaves a gap.

Phase	Method	Tool	Catches
1	Static analysis	Slither	Known patterns, code quality
2	Pattern detection	Aderyn	Domain-specific anti-patterns
3	Fuzz testing	Foundry	Implementation bugs, edge cases
4	Symbolic execution	Halmos	Logic bugs, invariant violations
5	Manual review	Human	Business logic, economic attacks

Table 1: The five-phase audit methodology.

1.3 Severity Classification

Severity	Count	Criteria
Critical	3	Direct fund loss, exploitable without special conditions
High	18	Indirect fund loss, requires specific conditions
Medium	37	Protocol malfunction, no direct fund loss
Low	41	Code quality, gas optimization, best practices
Informational	23	Style, documentation, non-functional
Total	**122**	

Table 2: Findings by severity across 50 contract domains.

2 Phase 1: Static Analysis with Slither

2.1 What Slither Does

Slither is a static analysis framework for Solidity. It parses Solidity source code into an intermediate representation (SlithIR) and runs detectors—pattern-matching rules that identify known vulnerability classes.

2.2 Running Slither

Listing 1: Running Slither on a Foundry project

```
# Install
pip install slither-analyzer

# Run all detectors
slither . --filter-paths "test/,script/,lib/"

# Run specific detector
```

```
slither . --detect reentrancy-eth

# Generate human-readable report
slither . --print human-summary

# Generate machine-readable JSON
slither . --json slither-report.json
```

2.3 Key Detectors

Detector	Vulnerability	Findings
`reentrancy-eth`	Reentrancy with ETH transfer	2
`reentrancy-no-eth`	Reentrancy without ETH transfer	5
`uninitialized-state`	Uninitialized storage variables	3
`arbitrary-send-eth`	Unchecked ETH transfers	4
`controlled-delegatecall`	Delegatecall with user input	1
`suicidal`	Unprotected selfdestruct	0
`unchecked-transfer`	Unchecked ERC-20 transfer	7
`locked-ether`	Contracts that receive but cannot send ETH	2

Table 3: Slither detector results across Lux Network contracts.

2.4 False Positives

Slither produces false positives. A finding is not a bug until confirmed by manual review. Our false positive rate across 50 domains: 34%. This means 66% of Slither findings are real issues. The remaining 34% are either intentional patterns (e.g., reentrancy guards that Slither does not recognize) or context-dependent non-issues.

3 Phase 2: Pattern Detection with Aderyn

Aderyn is a Rust-based Solidity analyzer that detects domain-specific anti-patterns that Slither misses. It is faster than Slither (Rust vs. Python) and focuses on patterns specific to DeFi, bridges, and governance.

Listing 2: Running Aderyn

```
# Install
cargo install aderyn

# Run analysis
aderyn .

# Output: report.md with findings by severity
```

Key patterns Aderyn detects that Slither misses:

- Centralization risks (single admin key controls protocol).
- Missing zero-address validation in constructors.
- Unsafe use of `block.timestamp` for time-sensitive logic.
- Missing event emissions for state changes.
- Inconsistent use of `SafeERC20`.

4 Phase 3: Fuzz Testing with Foundry

4.1 What Fuzzing Catches

Fuzzing generates random inputs and checks that invariants hold. It catches:

- Arithmetic overflow/underflow (despite Solidity 0.8 checks).
- Edge cases: zero amounts, maximum uint256, empty arrays.
- State-dependent bugs: invariant violations after specific transaction sequences.

4.2 Writing Fuzz Tests

Listing 3: Foundry fuzz test example

```
function testFuzz_depositWithdraw(
    uint256 amount,
    address user
) public {
    // Bound inputs to valid ranges
    amount = bound(amount, 1, 1e24);
    vm.assume(user != address(0));
    vm.assume(user != address(vault));

    // Setup
    deal(address(token), user, amount);
    vm.startPrank(user);
    token.approve(address(vault), amount);

    // Record state
    uint256 totalBefore = vault.totalAssets();

    // Deposit
    uint256 shares = vault.deposit(amount, user);
    assertGt(shares, 0, "shares must be nonzero");

    // Withdraw
    uint256 withdrawn = vault.redeem(shares, user, user);
    vm.stopPrank();

    // Invariant: withdrawn <= deposited (rounding loss ok)
    assertLe(withdrawn, amount, "cannot withdraw more than deposited");
    // Invariant: vault total assets are consistent
    assertEq(vault.totalAssets(), totalBefore, "total assets unchanged"
        );
}
```

4.3 Stateful Fuzzing (Invariant Testing)

Foundry's invariant testing mode generates random sequences of function calls and checks invariants after each call:

Listing 4: Stateful invariant test

```
// Handler contract: defines valid actions
contract PoolHandler is Test {
    Pool public pool;

    function swap(uint256 amount, bool direction) external {
        amount = bound(amount, 1, pool.reserve0() / 2);
```

```
        if (direction) {
            pool.swap(amount, 0, address(this));
        } else {
            pool.swap(0, amount, address(this));
        }
    }

    function addLiquidity(uint256 amount0, uint256 amount1) external {
        amount0 = bound(amount0, 1, 1e24);
        amount1 = bound(amount1, 1, 1e24);
        pool.addLiquidity(amount0, amount1, address(this));
    }
}

// Invariant contract
contract PoolInvariantTest is Test {
    PoolHandler handler;
    Pool pool;

    function setUp() public { /* ... */ }

    // This runs after every random action sequence
    function invariant_constantProduct() public {
        assertGe(
            pool.reserve0() * pool.reserve1(),
            initialK,
            "k must never decrease"
        );
    }

    function invariant_noFreeTokens() public {
        assertEq(
            token0.balanceOf(address(pool)),
            pool.reserve0(),
            "reserve0 matches actual balance"
        );
    }
}
```

4.4 Fuzzing Configuration

Listing 5: foundry.toml fuzz configuration

```
[fuzz]
runs = 10000            # Number of fuzz runs per test
max_test_rejects = 500 # Max invalid inputs before failing
seed = 0                # 0 = random seed each run
dictionary_weight = 40 # Weight for dictionary-based inputs

[invariant]
runs = 256              # Number of invariant test runs
depth = 128             # Max actions per run
fail_on_revert = false # Don't fail on expected reverts
```

5 Phase 4: Symbolic Execution with Halmos

Symbolic execution (covered in depth in our companion paper) proves properties for *all* inputs. In the audit context, we write Halmos tests for the most critical invariants:

- Token supply conservation.
- Bridge balance invariant (locked = minted on destination).
- Access control completeness (all admin functions revert for non-admins).
- Vault share/asset ratio monotonicity.

6 Phase 5: Manual Review

Automated tools catch *known* vulnerability patterns. Manual review catches:

- **Business logic errors**: The code does what it says, but what it says is wrong. Example: a fee calculation that charges 0.3% on the output instead of the input, giving arbitrageurs a profitable loop.
- **Economic attacks**: Flash loan attacks, oracle manipulation, sandwich attacks. These exploit *interactions between contracts* that automated tools analyze in isolation.
- **Governance attacks**: Vote buying, proposal frontrunning, timelock bypass.
- **Upgrade risks**: Proxy storage collisions, initialization order dependencies.

6.1 Manual Review Checklist

1. Read every line of code. Not skim—read.
2. For each external call: can the callee re-enter? What state is inconsistent at the call point?
3. For each `msg.sender` check: can it be bypassed via delegatecall or proxy?
4. For each arithmetic operation: what happens at the extremes (0, 1, type(uint256).max)?
5. For each oracle read: what if the oracle returns stale data? Zero? Negative?
6. For each token interaction: does it handle fee-on-transfer tokens? Rebasing tokens?
7. For each access control: is it consistent across all functions? Can admin keys be rotated?
8. For each storage write: is the order correct? Can it be frontrun?

7 Vulnerability Taxonomy

7.1 Reentrancy

Definition 1 (Reentrancy)**.** *A reentrancy vulnerability exists when a contract makes an external call before updating its state, allowing the callee to call back into the contract and observe or modify inconsistent state.*

Listing 6: Reentrancy vulnerability and fix

```
// VULNERABLE
function withdraw(uint256 amount) external {
    require(balances[msg.sender] >= amount);
    (bool ok,) = msg.sender.call{value: amount}("");
    require(ok);
    balances[msg.sender] -= amount; // Too late!
}

// FIXED: checks-effects-interactions pattern
function withdraw(uint256 amount) external {
    require(balances[msg.sender] >= amount);
    balances[msg.sender] -= amount; // State update first
    (bool ok,) = msg.sender.call{value: amount}("");
```

```
    require(ok);
}
```

Detection: Slither `reentrancy-eth`, manual review.

7.2 Flash Loan Attacks

Flash loans allow borrowing unlimited capital for the duration of a single transaction. This enables attacks on any protocol that relies on spot prices or instantaneous balances.

Listing 7: Flash loan attack on a naive oracle

```
// VULNERABLE: uses spot price
function getPrice() public view returns (uint256) {
    return reserve0 * 1e18 / reserve1; // Manipulable!
}

// FIXED: uses TWAP (time-weighted average price)
function getPrice() public view returns (uint256) {
    (uint256 price, uint256 lastUpdate) = oracle.consult(
        address(token), 1e18, 1800 // 30-minute TWAP
    );
    require(block.timestamp - lastUpdate < 3600, "stale");
    return price;
}
```

Detection: manual review (automated tools cannot reason about cross-contract economic interactions).

7.3 Oracle Manipulation

Listing 8: Oracle manipulation defense

```
// VULNERABLE: single oracle, no staleness check
function liquidate(address user) external {
    uint256 price = oracle.latestAnswer();
    require(debt(user) * price > collateral(user) * 1e18);
    // liquidate...
}

// FIXED: multiple oracles, staleness check, deviation check
function liquidate(address user) external {
    (uint256 price1, uint256 ts1) = oracle1.latestRoundData();
    (uint256 price2, uint256 ts2) = oracle2.latestRoundData();
    require(block.timestamp - ts1 < 3600, "oracle1 stale");
    require(block.timestamp - ts2 < 3600, "oracle2 stale");
    require(
        diff(price1, price2) * 100 / price1 < 5,
        "oracle deviation"
    );
    uint256 price = (price1 + price2) / 2;
    require(debt(user) * price > collateral(user) * 1e18);
    // liquidate...
}
```

7.4 Access Control

Listing 9: Access control vulnerability and fix

```
// VULNERABLE: missing access control
function setFee(uint256 newFee) external {
    fee = newFee; // Anyone can call this!
}

// FIXED: proper access control
function setFee(uint256 newFee) external onlyOwner {
    require(newFee <= MAX_FEE, "fee too high");
    emit FeeUpdated(fee, newFee);
    fee = newFee;
}
```

Detection: Slither `missing-zero-check`, Aderyn centralization detector.

7.5 Integer Overflow

Solidity 0.8+ has built-in overflow checks, but `unchecked` blocks bypass them:

Listing 10: Overflow in unchecked block

```
// VULNERABLE: overflow in unchecked block
function accumulateFees(uint256 newFees) internal {
    unchecked {
        totalFees += newFees; // Can wrap around!
    }
}

// FIXED: only use unchecked where overflow is proven impossible
function accumulateFees(uint256 newFees) internal {
    totalFees += newFees; // Solidity 0.8 overflow check
}
```

7.6 Frontrunning

Listing 11: Frontrunning defense with commit-reveal

```
// VULNERABLE: swap with no slippage protection
function swap(uint256 amountIn) external returns (uint256) {
    return _swap(amountIn); // Sandwich-able
}

// FIXED: minimum output amount
function swap(
    uint256 amountIn,
    uint256 minAmountOut,
    uint256 deadline
) external returns (uint256 amountOut) {
    require(block.timestamp <= deadline, "expired");
    amountOut = _swap(amountIn);
    require(amountOut >= minAmountOut, "slippage");
}
```

8 Test Coverage

Domain	Contracts	Tests	Branch %	Findings
AMM/DEX	8	147	96%	14
Bridge (Teleport)	6	98	94%	18
Yield vaults	5	72	91%	11
Staking	4	63	93%	8
Governance	7	85	89%	15
Oracle	3	41	95%	7
Token (ERC-20/721)	6	94	97%	12
Lending	4	67	90%	13
Other (33 domains)	45	165	88%	24
Total	**88**	**832**	**92%**	**122**

Table 4: Test coverage and findings across 50 contract domains.

9 CI Integration

Every pull request to a Lux Network smart contract repository runs the full audit pipeline:

Listing 12: CI audit pipeline

```
audit:
  steps:
    - name: Build
      run: forge build --sizes
    - name: Unit tests
      run: forge test -vvv
    - name: Coverage
      run: forge coverage --report lcov
    - name: Slither
      run: slither . --filter-paths "test/,script/,lib/"
             --fail-on medium
    - name: Aderyn
      run: aderyn .
    - name: Halmos
      run: halmos --function check_ --loop 5
    - name: Gas snapshot
      run: forge snapshot --check
```

The pipeline fails if:

- Any test fails.
- Branch coverage drops below 85%.
- Slither reports a medium-or-higher finding.
- Any Halmos symbolic test fails.
- Gas usage increases by more than 10% from the snapshot.

10 Conclusion

Smart contract auditing is not a single technique—it is a pipeline. Static analysis catches known patterns. Fuzzing catches implementation edge cases. Symbolic execution proves invariants. Manual review catches business logic errors. Each phase catches bugs that the others miss.

Our methodology, applied to 50 contract domains with 832 tests, found 122 issues including 3 critical vulnerabilities that would have resulted in direct fund loss. All findings were fixed

before deployment. The entire pipeline runs automatically on every pull request, ensuring that new code does not introduce regressions.

The most important lesson: no tool replaces reading the code. Tools find *known* bug patterns efficiently. Humans find *unknown* bug patterns—the ones that matter most.

References

[1] Feist, J. et al. "Slither: A Static Analysis Framework For Smart Contracts." WOOT, 2019.

[2] Paradigm. "Foundry: A Blazing Fast, Portable and Modular Toolkit for Ethereum Application Development." 2022.

[3] Park, D. et al. "Halmos: Symbolic Testing for EVM Smart Contracts." a16z Crypto, 2023.

[4] Cyfrin. "Aderyn: Rust-Based Solidity AST Analyzer." 2024.

[5] NCC Group. "Decentralized Application Security Project (DASP) Top 10." 2018.

[6] SmartContractSecurity. "SWC Registry: Smart Contract Weakness Classification." 2020.