/cairo-vulnerability-scanner

Source: `~/.claude/skills/tob-building-secure-contracts/skills/cairo-vulnerability-scanner/SKILL.md`

---

name: cairo-vulnerability-scanner
description: Scans Cairo/StarkNet smart contracts for 6 critical vulnerabilities including felt252 arithmetic overflow, L1-L2 messaging issues, address conversion problems, and signature replay. Use when auditing StarkNet projects.

Cairo/StarkNet Vulnerability Scanner

1. Purpose

Systematically scan Cairo smart contracts on StarkNet for platform-specific security vulnerabilities related to arithmetic, cross-layer messaging, and cryptographic operations. This skill encodes 6 critical vulnerability patterns unique to Cairo/StarkNet ecosystem.

2. When to Use This Skill

- Auditing StarkNet smart contracts (Cairo)
- Reviewing L1-L2 bridge implementations
- Pre-launch security assessment of StarkNet applications
- Validating cross-layer message handling
- Reviewing signature verification logic
- Assessing L1 handler functions

3. Platform Detection

File Extensions & Indicators

- **Cairo files**: `.cairo`

Language/Framework Markers

```
// Cairo contract indicators
#[contract]
mod MyContract {
    use starknet::ContractAddress;

    #[storage]
    struct Storage {
        balance: LegacyMap<ContractAddress, felt252>,
    }

    #[external(v0)]
    fn transfer(ref self: ContractState, to: ContractAddress, amount: felt252) {
        // Contract logic
    }
```

```
    #[l1_handler]
    fn handle_deposit(ref self: ContractState, from_address: felt252, amount: u256) {
        // L1 message handler
    }
}

// Common patterns
felt252, u128, u256
ContractAddress, EthAddress
#[external(v0)], #[l1_handler], #[constructor]
get_caller_address(), get_contract_address()
send_message_to_l1_syscall
```

Project Structure

- `src/contract.cairo` - Main contract implementation
- `src/lib.cairo` - Library modules
- `tests/` - Contract tests
- `Scarb.toml` - Cairo project configuration

Tool Support

- **Caracal**: Trail of Bits static analyzer for Cairo
- Installation: `pip install caracal`
- Usage: `caracal detect src/`
- **cairo-test**: Built-in testing framework
- **Starknet Foundry**: Testing and development toolkit

---

4. How This Skill Works

When invoked, I will:

1. **Search your codebase** for Cairo files
2. **Analyze each contract** for the 6 vulnerability patterns
3. **Report findings** with file references and severity
4. **Provide fixes** for each identified issue
5. **Check L1-L2 interactions** for messaging vulnerabilities

---

5. Example Output

When vulnerabilities are found, you'll get a report like this:

```
=== CAIRO/STARKNET VULNERABILITY SCAN RESULTS ===



---

## 5. Vulnerability Patterns (6 Patterns)

I check for 6 critical vulnerability patterns unique to Cairo/Starknet. For detailed detection
patterns, code examples, mitigations, and testing strategies, see
[VULNERABILITY_PATTERNS.md](resources/VULNERABILITY_PATTERNS.md).

### Pattern Summary:

1. **Unchecked Arithmetic** ⚠ CRITICAL - Integer overflow/underflow in felt252
2. **Storage Collision** ⚠ CRITICAL - Conflicting storage variable hashes
3. **Missing Access Control** ⚠ CRITICAL - No caller validation on sensitive functions
4. **Improper Felt252 Boundaries** ⚠ HIGH - Not validating felt252 range
5. **Unvalidated Contract Address** ⚠ HIGH - Using untrusted contract addresses
6. **Missing Caller Validation** ⚠ CRITICAL - No get_caller_address() checks

For complete vulnerability patterns with code examples, see
[VULNERABILITY_PATTERNS.md](resources/VULNERABILITY_PATTERNS.md).
## 5. Scanning Workflow

### Step 1: Platform Identification
1. Verify Cairo language and StarkNet framework
2. Check Cairo version (Cairo 1.0+ vs legacy Cairo 0)
3. Locate contract files (`src/*.cairo`)
4. Identify L1-L2 bridge contracts (if applicable)

### Step 2: Arithmetic Safety Sweep
```bash
# Find felt252 usage in arithmetic
rg "felt252" src/ | rg "[-+*/]"
```
```

```
# Find balance/amount storage using felt252
rg "felt252" src/ | rg "balance|amount|total|supply"

# Should prefer u128, u256 instead
```

Step 3: L1 Handler Analysis

For each `#[l1_handler]` function:

- [ ] Validates `from_address` parameter
- [ ] Checks address != zero
- [ ] Has proper access control
- [ ] Emits events for monitoring

Step 4: Signature Verification Review

For signature-based functions:

- [ ] Includes nonce tracking
- [ ] Nonce incremented after use
- [ ] Domain separator includes chain ID and contract address
- [ ] Cannot replay signatures

Step 5: L1-L2 Bridge Audit

If contract includes bridge functionality:

- [ ] L1 validates address < STARKNET_FIELD_PRIME
- [ ] L1 implements message cancellation
- [ ] L2 validates from_address in handlers
- [ ] Symmetric access controls L1 ↔ L2
- [ ] Test full roundtrip flows

Step 6: Static Analysis with Caracal

```
# Run Caracal detectors
caracal detect src/
```

```
# Specific detectors
caracal detect src/ --detectors unchecked-felt252-arithmetic
caracal detect src/ --detectors unchecked-l1-handler-from
caracal detect src/ --detectors missing-nonce-validation
```

---

6. Reporting Format

Finding Template

````
## [CRITICAL] Unchecked from_address in L1 Handler

**Location**: `src/bridge.cairo:145-155` (handle_deposit function)

**Description**:
The `handle_deposit` L1 handler function does not validate the `from_address` parameter. Any L1 contract can send messages to this function and mint tokens for arbitrary users, bypassing the intended L1 bridge access controls.

**Vulnerable Code**:
```rust
// bridge.cairo, line 145
#[l1_handler]
fn handle_deposit(
    ref self: ContractState,
    from_address: felt252,  // Not validated!
    user: ContractAddress,
    amount: u256
) {
    let current_balance = self.balances.read(user);
    self.balances.write(user, current_balance + amount);
}
````

Attack Scenario:

1. Attacker deploys malicious L1 contract
2. Malicious contract calls `starknetCore.sendMessageToL2(l2Contract, selector, [attacker_address, 1000000])`
````

3. L2 handler processes message without checking sender
4. Attacker receives 1,000,000 tokens without depositing any funds
5. Protocol suffers infinite mint vulnerability

Recommendation: Validate `from_address` against authorized L1 bridge:

```
#[l1_handler]
fn handle_deposit(
    ref self: ContractState,
    from_address: felt252,
    user: ContractAddress,
    amount: u256
) {
    // Validate L1 sender
    let authorized_l1_bridge = self.l1_bridge_address.read();
    assert(from_address == authorized_l1_bridge, 'Unauthorized L1 sender');

    let current_balance = self.balances.read(user);
    self.balances.write(user, current_balance + amount);
}
```

References:

- building-secure-contracts/not-so-smart-contracts/cairo/unchecked_l1_handler_from
- Caracal detector: `unchecked-l1-handler-from`

```
---

## 7. Priority Guidelines

### Critical (Immediate Fix Required)
- Unchecked from_address in L1 handlers (infinite mint)
- L1-L2 address conversion issues (funds to zero address)

### High (Fix Before Deployment)
- Felt252 arithmetic overflow/underflow (balance manipulation)
- Missing signature replay protection (replay attacks)
- L1-L2 message failure without cancellation (locked funds)

### Medium (Address in Audit)
```

```
- Overconstrained L1-L2 interactions (trapped funds)

---

## 8. Testing Recommendations

### Unit Tests
```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_felt252_overflow() {
        // Test arithmetic edge cases
    }

    #[test]
    #[should_panic]
    fn test_unauthorized_l1_handler() {
        // Wrong from_address should fail
    }

    #[test]
    fn test_signature_replay_protection() {
        // Same signature twice should fail
    }
}
```

Integration Tests (with L1)

```
// Test full L1-L2 flow
#[test]
fn test_deposit_withdraw_roundtrip() {
    // 1. Deposit on L1
    // 2. Wait for L2 processing
    // 3. Verify L2 balance
    // 4. Withdraw to L1
    // 5. Verify L1 balance restored
```
```

```
}
```

Caracal CI Integration

```
# .github/workflows/security.yml
- name: Run Caracal
  run: |
    pip install caracal
    caracal detect src/ --fail-on high,critical
```

---

9. Additional Resources

- **Building Secure Contracts**: `building-secure-contracts/not-so-smart-contracts/cairo/`
- **Caracal**: https://github.com/crytic/caracal
- **Cairo Documentation**: https://book.cairo-lang.org/
- **StarkNet Documentation**: https://docs.starknet.io/
- **OpenZeppelin Cairo Contracts**: https://github.com/OpenZeppelin/cairo-contracts

---

10. Quick Reference Checklist

Before completing Cairo/StarkNet audit:

Arithmetic Safety (HIGH):

- ☐ No felt252 used for balances/amounts (use u128/u256)
- ☐ OR felt252 arithmetic has explicit bounds checking
- ☐ Overflow/underflow scenarios tested

L1 Handler Security (CRITICAL):

- ☐ ALL `#[l1_handler]` functions validate `from_address`
- ☐ from_address compared against stored L1 contract address
- ☐ Cannot bypass by deploying alternate L1 contract

L1-L2 Messaging (HIGH):

- ☐ L1 bridge validates addresses < STARKNET_FIELD_PRIME

- [ ] L1 bridge implements message cancellation
- [ ] L2 handlers check from_address
- [ ] Symmetric validation rules L1 ↔ L2
- [ ] Full roundtrip flows tested

Signature Security (HIGH):

- [ ] Signatures include nonce tracking
- [ ] Nonce incremented after each use
- [ ] Domain separator includes chain ID and contract address
- [ ] Signature replay tested and prevented
- [ ] Cross-chain replay prevented

Tool Usage:

- [ ] Caracal scan completed with no critical findings
- [ ] Unit tests cover all vulnerability scenarios
- [ ] Integration tests verify L1-L2 flows
- [ ] Testnet deployment tested before mainnet

Revision #4
Created 2026-02-18 08:40:03 UTC by John
Updated 2026-05-31 20:01:37 UTC by John