/algorand-vulnerability-scanner

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

---

name: algorand-vulnerability-scanner
description: Scans Algorand smart contracts for 11 common vulnerabilities including rekeying attacks, unchecked transaction fees, missing field validations, and access control issues. Use when auditing Algorand projects (TEAL/PyTeal).

Algorand Vulnerability Scanner

1. Purpose

Systematically scan Algorand smart contracts (TEAL and PyTeal) for platform-specific security vulnerabilities documented in Trail of Bits' "Not So Smart Contracts" database. This skill encodes 11 critical vulnerability patterns unique to Algorand's transaction model.

2. When to Use This Skill

- Auditing Algorand smart contracts (stateful applications or smart signatures)
- Reviewing TEAL assembly or PyTeal code
- Pre-audit security assessment of Algorand projects
- Validating fixes for reported Algorand vulnerabilities
- Training team on Algorand-specific security patterns

3. Platform Detection

File Extensions & Indicators

- **TEAL files**: `.teal`
- **PyTeal files**: `.py` with PyTeal imports

Language/Framework Markers

```
# PyTeal indicators
from pyteal import *
from algosdk import *

# Common patterns
Txn, Gtxn, Global, InnerTxnBuilder
OnComplete, ApplicationCall, TxnType
@router.method, @Subroutine
```

Project Structure

- `approval_program.py` / `clear_program.py`
- `contract.teal` / `signature.teal`
- References to Algorand SDK or Beaker framework

Tool Support

- **Tealer**: Trail of Bits static analyzer for Algorand
- Installation: `pip3 install tealer`
- Usage: `tealer contract.teal --detect all`

---

4. How This Skill Works

When invoked, I will:

1. **Search your codebase** for TEAL/PyTeal files
2. **Analyze each file** for the 11 vulnerability patterns
3. **Report findings** with file references and severity
4. **Provide fixes** for each identified issue
5. **Run Tealer** (if installed) for automated detection

---

5. Example Output

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

```
=== ALGORAND VULNERABILITY SCAN RESULTS ===

Project: my-algorand-dapp
Files Scanned: 3 (.teal, .py)
Vulnerabilities Found: 2

---

[CRITICAL] Rekeying Attack
File: contracts/approval.py:45
Pattern: Missing RekeyTo validation

Code:
    If(Txn.type_enum() == TxnType.Payment,
        Seq([
            # Missing: Assert(Txn.rekey_to() == Global.zero_address())
            App.globalPut(Bytes("balance"), balance + Txn.amount()),
            Approve()
        ])
```

```
    )

Issue: The contract doesn't validate the RekeyTo field, allowing attackers
to change account authorization and bypass restrictions.


---

## 5. Vulnerability Patterns (11 Patterns)

I check for 11 critical vulnerability patterns unique to Algorand. For detailed detection
patterns, code examples, mitigations, and testing strategies, see
[VULNERABILITY_PATTERNS.md](resources/VULNERABILITY_PATTERNS.md).

### Pattern Summary:

1. **Rekeying Vulnerability** ⚠ CRITICAL - Unchecked RekeyTo field
2. **Missing Transaction Verification** ⚠ CRITICAL - No GroupSize/GroupIndex checks
3. **Group Transaction Manipulation** ⚠ HIGH - Unsafe group transaction handling
4. **Asset Clawback Risk** ⚠ HIGH - Missing clawback address checks
5. **Application State Manipulation** ⚠ MEDIUM - Unsafe global/local state updates
6. **Asset Opt-In Missing** ⚠ HIGH - No asset opt-in validation
7. **Minimum Balance Violation** ⚠ MEDIUM - Account below minimum balance
8. **Close Remainder To Check** ⚠ HIGH - Unchecked CloseRemainderTo field
9. **Application Clear State** ⚠ MEDIUM - Unsafe clear state program
10. **Atomic Transaction Ordering** ⚠ HIGH - Assuming transaction order
11. **Logic Signature Reuse** ⚠ HIGH - Logic sigs without uniqueness constraints

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

### Step 1: Platform Identification
1. Confirm file extensions (`.teal`, `.py`)
2. Identify framework (PyTeal, Beaker, pure TEAL)
3. Determine contract type (stateful application vs smart signature)
4. Locate approval and clear state programs

### Step 2: Static Analysis with Tealer
```

```bash
# Run Tealer on contract
tealer contract.teal --detect all

# Or specific detectors
tealer contract.teal --detect unprotected-rekey,group-size-check,update-application-check
```

Step 3: Manual Vulnerability Sweep

For each of the 11 vulnerabilities above:

1. Search for relevant transaction field usage
2. Verify validation logic exists
3. Check for bypass conditions
4. Validate inner transaction handling

Step 4: Transaction Field Validation Matrix

Create checklist for all transaction types used:

Payment Transactions:

- ☐ RekeyTo validated
- ☐ CloseRemainderTo validated
- ☐ Fee validated (if smart signature)

Asset Transfers:

- ☐ Asset ID validated
- ☐ AssetCloseTo validated
- ☐ RekeyTo validated

Application Calls:

- ☐ OnComplete validated
- ☐ Access controls enforced
- ☐ Group size validated

Inner Transactions:

- ☐ Fee explicitly set to 0

- [ ] RekeyTo not user-controlled (Teal v6+)
- [ ] All fields validated

Step 5: Group Transaction Analysis

For atomic transaction groups:

1. Validate `Global.group_size()` checks
2. Review absolute vs relative indexing
3. Check for replay protection (Lease field)
4. Verify OnComplete fields for ApplicationCalls in group

Step 6: Access Control Review

- [ ] Creator/admin privileges properly enforced
- [ ] Update/delete operations protected
- [ ] Sensitive functions have authorization checks

---

6. Reporting Format

Finding Template

````
## [SEVERITY] Vulnerability Name (e.g., Missing RekeyTo Validation)

**Location**: `contract.teal:45-50` or `approval_program.py:withdraw()`

**Description**:
The contract approves payment transactions without validating the RekeyTo field, allowing an
attacker to rekey the account and bypass future authorization checks.

**Vulnerable Code**:
```python
# approval_program.py, line 45
If(Txn.type_enum() == TxnType.Payment,
    Approve()  # Missing RekeyTo check
)
````
````

Attack Scenario:

1. Attacker submits payment transaction with RekeyTo set to attacker's address
2. Contract approves transaction without checking RekeyTo
3. Account authorization is rekeyed to attacker
4. Attacker gains full control of account

Recommendation: Add explicit validation of the RekeyTo field:

```
If(And(
    Txn.type_enum() == TxnType.Payment,
    Txn.rekey_to() == Global.zero_address()
), Approve(), Reject())
```

References:

- building-secure-contracts/not-so-smart-contracts/algorand/rekeying
- Tealer detector: `unprotected-rekey`

```
---

## 7. Priority Guidelines

### Critical (Immediate Fix Required)
- Rekeying attacks
- CloseRemainderTo / AssetCloseTo issues
- Access control bypasses

### High (Fix Before Deployment)
- Unchecked transaction fees
- Asset ID validation issues
- Group size validation
- Clear state transaction checks

### Medium (Address in Audit)
- Inner transaction fee issues
- Time-based replay attacks
- DoS via asset opt-in

---
```

```
## 8. Testing Recommendations

### Unit Tests Required
- Test each vulnerability scenario with PoC exploit
- Verify fixes prevent exploitation
- Test edge cases (group size = 0, empty addresses, etc.)

### Tealer Integration
```bash
# Add to CI/CD pipeline
tealer approval.teal --detect all --json > tealer-report.json

# Fail build on critical findings
tealer approval.teal --detect all --fail-on critical,high
```

Scenario Testing

- Submit transactions with all critical fields manipulated
- Test atomic groups with unexpected sizes
- Attempt access control bypasses
- Verify inner transaction fee handling

---

9. Additional Resources

- **Building Secure Contracts**: `building-secure-contracts/not-so-smart-contracts/algorand/`
- **Tealer Documentation**: https://github.com/crytic/tealer
- **Algorand Developer Docs**: https://developer.algorand.org/docs/
- **PyTeal Documentation**: https://pyteal.readthedocs.io/

---

10. Quick Reference Checklist

Before completing Algorand audit, verify ALL items checked:

- ☐ RekeyTo validated in all transaction types
- ☐ CloseRemainderTo validated in payment transactions
- ☐ AssetCloseTo validated in asset transfers
```

- [ ] Transaction fees validated (smart signatures)
- [ ] Group size validated for atomic transactions
- [ ] Lease field used for replay protection (where applicable)
- [ ] Access controls on Update/Delete operations
- [ ] Asset ID validated in all asset operations
- [ ] Asset transfers use pull pattern to avoid DoS
- [ ] Inner transaction fees explicitly set to 0
- [ ] OnComplete field validated for ApplicationCall transactions
- [ ] Tealer scan completed with no critical/high findings
- [ ] Unit tests cover all vulnerability scenarios

Revision #4
Created 2026-02-18 08:40:02 UTC by John
Updated 2026-05-31 20:01:36 UTC by John