/entry-point-analyzer

Source: `~/.claude/skills/tob-entry-point-analyzer/skills/entry-point-analyzer/SKILL.md`

---

name: entry-point-analyzer description: Analyzes smart contract codebases to identify state-changing entry points for security auditing. Detects externally callable functions that modify state, categorizes them by access level (public, admin, role-restricted, contract-only), and generates structured audit reports. Excludes view/pure/read-only functions. Use when auditing smart contracts (Solidity, Vyper, Solana/Rust, Move, TON, CosmWasm) or when asked to find entry points, audit flows, external functions, access control patterns, or privileged operations. allowed-tools:

- Read
- Grep
- Glob
- Bash

Entry Point Analyzer

Systematically identify all **state-changing** entry points in a smart contract codebase to guide security audits.

When to Use

Use this skill when:

- Starting a smart contract security audit to map the attack surface
- Asked to find entry points, external functions, or audit flows
- Analyzing access control patterns across a codebase
- Identifying privileged operations and role-restricted functions
- Building an understanding of which functions can modify contract state

When NOT to Use

Do NOT use this skill for:

- Vulnerability detection (use audit-context-building or domain-specific-audits)
- Writing exploit POCs (use solidity-poc-builder)
- Code quality or gas optimization analysis
- Non-smart-contract codebases
- Analyzing read-only functions (this skill excludes them)

Scope: State-Changing Functions Only

This skill focuses exclusively on functions that can modify state. **Excluded:**

Language	Excluded Patterns
Solidity	`view`, `pure` functions
Vyper	`@view`, `@pure` functions
Solana	Functions without `mut` account references
Move	Non-entry `public fun` (module-callable only)
TON	`get` methods (FunC), read-only receivers (Tact)
CosmWasm	`query` entry point and its handlers

Why exclude read-only functions? They cannot directly cause loss of funds or state corruption. While they may leak information, the primary audit focus is on functions that can change state.

Workflow

1. **Detect Language** - Identify contract language(s) from file extensions and syntax
2. **Use Tooling (if available)** - For Solidity, check if Slither is available and use it
3. **Locate Contracts** - Find all contract/module files (apply directory filter if specified)
4. **Extract Entry Points** - Parse each file for externally callable, state-changing functions
5. **Classify Access** - Categorize each function by access level
6. **Generate Report** - Output structured markdown report

Slither Integration (Solidity)

For Solidity codebases, Slither can automatically extract entry points. Before manual analysis:

1. Check if Slither is Available

```
which slither
```

2. If Slither is Detected, Run Entry Points Printer

```
slither . --print entry-points
```

This outputs a table of all state-changing entry points with:

- Contract name
- Function name
- Visibility
- Modifiers applied

3. Use Slither Output as Foundation

- Parse the Slither output table to populate your analysis
- Cross-reference with manual inspection for access control classification
- Slither may miss some patterns (callbacks, dynamic access control)—supplement with manual review
- If Slither fails (compilation errors, unsupported features), fall back to manual analysis

4. When Slither is NOT Available

If `which slither` returns nothing, proceed with manual analysis using the language-specific reference files.

Language Detection

Extension	Language	Reference
`.sol`	Solidity	{baseDir}/references/solidity.md
`.vy`	Vyper	{baseDir}/references/vyper.md
`.rs` + `Cargo.toml` with `solana-program`	Solana (Rust)	{baseDir}/references/solana.md
`.move` + `Move.toml` with `edition`	{baseDir}/references/move-sui.md	
`.move` + `Move.toml` with `Aptos`	{baseDir}/references/move-aptos.md	

Extension	Language	Reference
`.fc`, `.func`, `.tact`	TON (FunC/Tact)	{baseDir}/references/ton.md
`.rs` + `Cargo.toml` with `cosmwasm-std`	CosmWasm	{baseDir}/references/cosmwasm.md

Load the appropriate reference file(s) based on detected language before analysis.

Access Classification

Classify each state-changing entry point into one of these categories:

1. Public (Unrestricted)

Functions callable by anyone without restrictions.

2. Role-Restricted

Functions limited to specific roles. Common patterns to detect:

- Explicit role names: `admin`, `owner`, `governance`, `guardian`, `operator`, `manager`, `minter`, `pauser`, `keeper`, `relayer`, `lender`, `borrower`
- Role-checking patterns: `onlyRole`, `hasRole`, `require(msg.sender == X)`, `assert_owner`, `#[access_control]`
- When role is ambiguous, flag as **"Restricted (review required)"** with the restriction pattern noted

3. Contract-Only (Internal Integration Points)

Functions callable only by other contracts, not by EOAs. Indicators:

- Callbacks: `onERC721Received`, `uniswapV3SwapCallback`, `flashLoanCallback`
- Interface implementations with contract-caller checks
- Functions that revert if `tx.origin == msg.sender`
- Cross-contract hooks

Output Format

Generate a markdown report with this structure:

```
# Entry Point Analysis: [Project Name]

**Analyzed**: [timestamp]
**Scope**: [directories analyzed or "full codebase"]
**Languages**: [detected languages]
**Focus**: State-changing functions only (view/pure excluded)

## Summary

| Category | Count |
|----------|-------|
| Public (Unrestricted) | X |
| Role-Restricted | X |
| Restricted (Review Required) | X |
| Contract-Only | X |
| **Total** | **X** |

---

## Public Entry Points (Unrestricted)

State-changing functions callable by anyone—prioritize for attack surface analysis.

| Function | File | Notes |
|----------|------|-------|
| `functionName(params)` | `path/to/file.sol:L42` | Brief note if relevant |

---

## Role-Restricted Entry Points

### Admin / Owner
| Function | File | Restriction |
|----------|------|-------------|
| `setFee(uint256)` | `Config.sol:L15` | `onlyOwner` |

### Governance
| Function | File | Restriction |
|----------|------|-------------|
```

```
### Guardian / Pauser
| Function | File | Restriction |
|----------|------|-------------|

### Other Roles
| Function | File | Restriction | Role |
|----------|------|-------------|------|

---

## Restricted (Review Required)

Functions with access control patterns that need manual verification.

| Function | File | Pattern | Why Review |
|----------|------|---------|------------|
| `execute(bytes)` | `Executor.sol:L88` | `require(trusted[msg.sender])` | Dynamic trust list
|

---

## Contract-Only (Internal Integration Points)

Functions only callable by other contracts—useful for understanding trust boundaries.

| Function | File | Expected Caller |
|----------|------|-----------------|
| `onFlashLoan(...)` | `Vault.sol:L200` | Flash loan provider |

---

## Files Analyzed

- `path/to/file1.sol` (X state-changing entry points)
- `path/to/file2.sol` (X state-changing entry points)
```

Filtering

When user specifies a directory filter:

- Only analyze files within that path
- Note the filter in the report header
- Example: "Analyze only `src/core/`" → scope = `src/core/`

Analysis Guidelines

1. **Be thorough**: Don't skip files. Every state-changing externally callable function matters.
2. **Be conservative**: When uncertain about access level, flag for review rather than miscategorize.
3. **Skip read-only**: Exclude `view`, `pure`, and equivalent read-only functions.
4. **Note inheritance**: If a function's access control comes from a parent contract, note this.
5. **Track modifiers**: List all access-related modifiers/decorators applied to each function.
6. **Identify patterns**: Look for common patterns like:
   - Initializer functions (often unrestricted on first call)
   - Upgrade functions (high-privilege)
   - Emergency/pause functions (guardian-level)
   - Fee/parameter setters (admin-level)
   - Token transfers and approvals (often public)

Common Role Patterns by Protocol Type

Protocol Type	Common Roles
DEX	`owner`, `feeManager`, `pairCreator`
Lending	`admin`, `guardian`, `liquidator`, `oracle`
Governance	`proposer`, `executor`, `canceller`, `timelock`
NFT	`minter`, `admin`, `royaltyReceiver`
Bridge	`relayer`, `guardian`, `validator`, `operator`
Vault/Yield	`strategist`, `keeper`, `harvester`, `manager`

Rationalizations to Reject

When analyzing entry points, reject these shortcuts:

- "This function looks standard" → Still classify it; standard functions can have non-standard access control
- "The modifier name is clear" → Verify the modifier's actual implementation
- "This is obviously admin-only" → Trace the actual restriction; "obvious" assumptions miss subtle bypasses
- "I'll skip the callbacks" → Callbacks define trust boundaries; always include them
- "It doesn't modify much state" → Any state change can be exploited; include all non-view functions

Error Handling

If a file cannot be parsed:

1. Note it in the report under "Analysis Warnings"
2. Continue with remaining files
3. Suggest manual review for unparsable files

Revision #4
Created 2026-02-18 08:40:06 UTC by John
Updated 2026-05-31 20:01:49 UTC by John