/substrate-vulnerability-scanner

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

---

name: substrate-vulnerability-scanner
description: Scans Substrate/Polkadot pallets for 7 critical vulnerabilities including arithmetic overflow, panic DoS, incorrect weights, and bad origin checks. Use when auditing Substrate runtimes or FRAME pallets.

Substrate Vulnerability Scanner

1. Purpose

Systematically scan Substrate runtime modules (pallets) for platform-specific security vulnerabilities that can cause node crashes, DoS attacks, or unauthorized access. This skill encodes

7 critical vulnerability patterns unique to Substrate/FRAME-based chains.

2. When to Use This Skill

- Auditing custom Substrate pallets
- Reviewing FRAME runtime code
- Pre-launch security assessment of Substrate chains (Polkadot parachains, standalone chains)
- Validating dispatchable extrinsic functions
- Reviewing weight calculation functions
- Assessing unsigned transaction validation logic

3. Platform Detection

File Extensions & Indicators

- **Rust files**: `.rs`

Language/Framework Markers

```
// Substrate/FRAME indicators
#[pallet]
pub mod pallet {
    use frame_support::pallet_prelude::*;
    use frame_system::pallet_prelude::*;

    #[pallet::config]
    pub trait Config: frame_system::Config { }

    #[pallet::call]
    impl<T: Config> Pallet<T> {
        #[pallet::weight(10_000)]
        pub fn example_function(origin: OriginFor<T>) -> DispatchResult { }
    }
}

// Common patterns
```

```
DispatchResult, DispatchError
ensure!, ensure_signed, ensure_root
StorageValue, StorageMap, StorageDoubleMap
#[pallet::storage]
#[pallet::call]
#[pallet::weight]
#[pallet::validate_unsigned]
```

Project Structure

- `pallets/*/lib.rs` - Pallet implementations
- `runtime/lib.rs` - Runtime configuration
- `benchmarking.rs` - Weight benchmarks
- `Cargo.toml` with `frame-*` dependencies

Tool Support

- **cargo-fuzz**: Fuzz testing for Rust
- **test-fuzz**: Property-based testing framework
- **benchmarking framework**: Built-in weight calculation
- **try-runtime**: Runtime migration testing

---

4. How This Skill Works

When invoked, I will:

1. **Search your codebase** for Substrate pallets
2. **Analyze each pallet** for the 7 vulnerability patterns
3. **Report findings** with file references and severity
4. **Provide fixes** for each identified issue
5. **Check weight calculations** and origin validation

---

5. Vulnerability Patterns (7 Critical Patterns)

I check for 7 critical vulnerability patterns unique to Substrate/FRAME. For detailed detection patterns, code examples, mitigations, and testing strategies, see VULNERABILITY_PATTERNS.md.

Pattern Summary:

1. **Arithmetic Overflow** ⚠ CRITICAL
   - Direct `+`, `-`, `*`, `/` operators wrap in release mode
   - Must use `checked_*` or `saturating_*` methods
   - Affects balance/token calculations, reward/fee math
2. **Don't Panic** ⚠ CRITICAL - DoS
   - Panics cause node to stop processing blocks
   - No `unwrap()`, `expect()`, array indexing without bounds check
   - All user input must be validated with `ensure!`
3. **Weights and Fees** ⚠ CRITICAL - DoS
   - Incorrect weights allow spam attacks
   - Fixed weights for variable-cost operations enable DoS
   - Must use benchmarking framework, bound all input parameters
4. **Verify First, Write Last** ⚠ HIGH (Pre-v0.9.25)
   - Storage writes before validation persist on error (pre-v0.9.25)
   - Pattern: validate → write → emit event
   - Upgrade to v0.9.25+ or use manual `#[transactional]`
5. **Unsigned Transaction Validation** ⚠ HIGH
   - Insufficient validation allows spam/replay attacks
   - Prefer signed transactions
   - If unsigned: validate parameters, replay protection, authenticate source
6. **Bad Randomness** ⚠ MEDIUM
   - `pallet_randomness_collective_flip` vulnerable to collusion
   - Must use BABE randomness (`pallet_babe::RandomnessFromOneEpochAgo`)
   - Use `random(subject)` not `random_seed()`
7. **Bad Origin** ⚠ CRITICAL
   - `ensure_signed` allows any user for privileged operations
   - Must use `ensure_root` or custom origins (ForceOrigin, AdminOrigin)
   - Origin types must be properly configured in runtime

For complete vulnerability patterns with code examples, see VULNERABILITY_PATTERNS.md.

---

6. Scanning Workflow

Step 1: Platform Identification

1. Verify Substrate/FRAME framework usage

2. Check Substrate version (v0.9.25+ has transactional storage)
3. Locate pallet implementations (`pallets/*/lib.rs`)
4. Identify runtime configuration (`runtime/lib.rs`)

Step 2: Dispatchable Analysis

For each `#[pallet::call]` function:

- [ ] Arithmetic: Uses checked/saturating operations?
- [ ] Panics: No unwrap/expect/indexing?
- [ ] Weights: Proportional to cost, bounded inputs?
- [ ] Origin: Appropriate validation level?
- [ ] Validation: All checks before storage writes?

Step 3: Panic Sweep

```
# Search for panic-prone patterns
rg "unwrap\(\)" pallets/
rg "expect\(" pallets/
rg "\[.*\]" pallets/  # Array indexing
rg " as u\d+" pallets/  # Type casts
rg "\.unwrap_or" pallets/
```

Step 4: Arithmetic Safety Check

```
# Find direct arithmetic
rg " \+ |\+=| - |-=| \* |\*=| / |/=" pallets/

# Should find checked/saturating alternatives instead
rg "checked_add|checked_sub|checked_mul|checked_div" pallets/
rg "saturating_add|saturating_sub|saturating_mul" pallets/
```

Step 5: Weight Analysis

- [ ] Run benchmarking: `cargo test --features runtime-benchmarks`
- [ ] Verify weights match computational cost
- [ ] Check for bounded input parameters

- [ ] Review weight calculation functions

Step 6: Origin & Privilege Review

```
# Find privileged operations
rg "ensure_signed" pallets/ | grep -E "pause|emergency|admin|force|sudo"

# Should use ensure_root or custom origins
rg "ensure_root|ForceOrigin|AdminOrigin" pallets/
```

Step 7: Testing Review

- [ ] Unit tests cover all dispatchables
- [ ] Fuzz tests for panic conditions
- [ ] Benchmarks for weight calculation
- [ ] try-runtime tests for migrations

---

7. Priority Guidelines

Critical (Immediate Fix Required)

- Arithmetic overflow (token creation, balance manipulation)
- Panic DoS (node crash risk)
- Bad origin (unauthorized privileged operations)

High (Fix Before Launch)

- Incorrect weights (DoS via spam)
- Verify-first violations (state corruption, pre-v0.9.25)
- Unsigned validation issues (spam, replay attacks)

Medium (Address in Audit)

- Bad randomness (manipulation possible but limited impact)

---

8. Testing Recommendations

Fuzz Testing

```
// Use test-fuzz for property-based testing
#[cfg(test)]
mod tests {
    use test_fuzz::test_fuzz;

    #[test_fuzz]
    fn fuzz_transfer(from: AccountId, to: AccountId, amount: u128) {
        // Should never panic
        let _ = Pallet::transfer(from, to, amount);
    }

    #[test_fuzz]
    fn fuzz_no_panics(call: Call) {
        // No dispatchable should panic
        let _ = call.dispatch(origin);
    }
}
```

Benchmarking

```
# Run benchmarks to generate weights
cargo build --release --features runtime-benchmarks
./target/release/node benchmark pallet \
    --chain dev \
    --pallet pallet_example \
    --extrinsic "*" \
    --steps 50 \
    --repeat 20
```

try-runtime

```
# Test runtime upgrades
cargo build --release --features try-runtime
```

```
try-runtime --runtime ./target/release/wbuild/runtime.wasm \
    on-runtime-upgrade live --uri wss://rpc.polkadot.io
```

---

9. Additional Resources

- **Building Secure Contracts**: `building-secure-contracts/not-so-smart-contracts/substrate/`
- **Substrate Documentation**: https://docs.substrate.io/
- **FRAME Documentation**: https://paritytech.github.io/substrate/master/frame_support/
- **test-fuzz**: https://github.com/trailofbits/test-fuzz
- **Substrate StackExchange**: https://substrate.stackexchange.com/

---

10. Quick Reference Checklist

Before completing Substrate pallet audit:

Arithmetic Safety (CRITICAL):

- ☐ No direct `+`, `-`, `*`, `/` operators in dispatchables
- ☐ All arithmetic uses `checked_*` or `saturating_*`
- ☐ Type conversions use `try_into()` with error handling

Panic Prevention (CRITICAL):

- ☐ No `unwrap()` or `expect()` in dispatchables
- ☐ No direct array/slice indexing without bounds check
- ☐ All user inputs validated with `ensure!`
- ☐ Division operations check for zero divisor

Weights & DoS (CRITICAL):

- ☐ Weights proportional to computational cost
- ☐ Input parameters have maximum bounds
- ☐ Benchmarking used to determine weights
- ☐ No free (zero-weight) expensive operations

Access Control (CRITICAL):

- [ ] Privileged operations use `ensure_root` or custom origins
- [ ] `ensure_signed` only for user-level operations
- [ ] Origin types properly configured in runtime
- [ ] Sudo pallet removed before production

Storage Safety (HIGH):

- [ ] Using Substrate v0.9.25+ OR manual `#[transactional]`
- [ ] Validation before storage writes
- [ ] Events emitted after successful operations

Other (MEDIUM):

- [ ] Unsigned transactions use signed alternative if possible
- [ ] If unsigned: proper validation, replay protection, authentication
- [ ] BABE randomness used (not RandomnessCollectiveFlip)
- [ ] Randomness uses `random(subject)` not `random_seed()`

Testing:

- [ ] Unit tests for all dispatchables
- [ ] Fuzz tests to find panics
- [ ] Benchmarks generated and verified
- [ ] try-runtime tests for migrations

Revision #4
Created 2026-02-18 08:40:04 UTC by John
Updated 2026-05-31 20:01:43 UTC by John