// HALBORN

Shift Markets - Cables DEX

Soroban Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **October 30th, 2023 - November 15th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Draft Version	11/15/2023
0.2	Draft Review	11/17/2023
1.0	Remediation Plan	11/29/2023
1.1	Remediation Plan Review	12/04/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

The Cables project is a decentralized exchange (DEX) smart contract system built on the Stellar blockchain using the Soroban framework. It facilitates noncustodial access to centralized liquidity by allowing users to place trades through smart contracts that store funds and trade data.

The DEX supports three order types -- market, limit, and OTC -- offering flexibility to users. Additionally, it employs a hybrid model where off-chain centralized order books mirror on-chain trades, ensuring efficient and secure execution. The system is designed to uphold the noncustodial nature of the protocol, providing users with control over their funds, while also supporting multiple execution facilities with customizable parameters, enhancing the overall trading experience.

Shift Markets engaged Halborn to conduct a security assessment on their smart contracts beginning on October 30th, 2023 and ending on November 15th, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team, specified in the Scope section.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided two weeks and a half for the engagement and assigned a full-time security engineer to verify the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some security risks that were successfully

addressed by the Shift Markets team.

(HAL-01) ZERO CHECK MISSING

The contract buy and sell functions allow the creation of a new order with a deposit amount of 0, facilitated by the transfer function of the Token contract. Due to a limitation in the check_nonnegative_amount function, which only checks for amounts less than 0, users can exploit these functions to stress the system with empty transactions, potentially causing a Denial of Service over time as the CurOrder counter key increments without overflow handling.

(HAL-02) LACK OF OVERFLOW CONTROL

The save_order function is responsible for storing information about a new order in the ledger, including the increment of the CurOrder counter. However, this counter lacks overflow control, and if it reaches its maximum value, the operation will panic, preventing the creation of new orders. To address this, it is advisable to use overflow-controlled operations like checked_add for counters, which returns None if the maximum is reached.

(HAL-03) OWNERSHIP CAN BE TRANSFERRED WITHOUT CONFIRMATION

The set_admin function permits changing the contract's owner/administrator, exclusively executable by the current administrator. However, unintended errors (e.g., an incorrect address) during transaction execution could result in a loss of control over the contract and its privileged functions. To enhance security, it is advisable to implement a two-step process: first, a transaction to set the new administrator value, and second, another transaction to confirm this information, signed by the new administrator, reducing the risk of inadvertent loss of control.

(HAL-04) CONTRACT ADMINISTRATOR NOT ALLOWED TO FILL SOME ORDERS

The fill function facilitates the exchange of assets by matching buy/sell orders, with access restricted to administrators. However, the contract administrator faces a limitation as they can only fill orders executed in the ADMIN execution facility. To maintain the administrator's privilege, the described checks should include an exception for it, otherwise, the access control for the contract administrator can be removed.

1.3 SCOPE

- Cables - Shift Markets:
  - Repository: Cables
  - Commit ID: 72e57e8
  - Smart contracts in scope:
    - soroban_cable_contract

Out-of-scope: External libraries and financial related attacks.

1.4 TEST APPROACH & METHODOLOGY

Halborn performed a combination of the manual view of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation, automated testing techniques help enhance the coverage of smart contracts. They can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into architecture, purpose, and use of the platform.
- Manual code read and walk through.
- Manual Assessment of use and safety for the critical Rust variables and functions in scope to identify any arithmetic related vulnerability classes.
- Cross contract call controls.
- Architecture related logical controls.
- Scanning of Rust files for vulnerabilities.(cargo audit)
- Deployment to local testnet using docker environment and interacting with Soroban CLI.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	2	1	6

EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) ZERO CHECK MISSING	Medium (6.7)	SOLVED - 11/22/2023
(HAL-02) LACK OF OVERFLOW CONTROL	Medium (6.7)	SOLVED - 11/29/2023
(HAL-03) OWNERSHIP CAN BE TRANSFERRED WITHOUT CONFIRMATION	Low (2.5)	SOLVED - 11/23/2023
(HAL-04) CONTRACT ADMINISTRATOR NOT ALLOWED TO FILL SOME ORDERS	Informational (1.7)	SOLVED - 11/29/2023
(HAL-05) ERROR HANDLING	Informational (1.7)	SOLVED - 11/29/2023
(HAL-06) LACK OF DEBUGGING INFO	Informational (1.7)	SOLVED - 11/24/2023
(HAL-07) MISUSE OF RUST HELPER METHODS	Informational (0.0)	SOLVED - 11/29/2023
(HAL-08) ASSIGNMENT NOT NEEDED	Informational (0.0)	SOLVED - 11/21/2023
(HAL-09) UPDATE CONTRACT AND FACILITY ADMINISTRATOR AT SAME TIME	Informational (0.0)	SOLVED - 11/23/2023

FINDINGS & TECH DETAILS

3.1 (HAL-01) ZERO CHECK MISSING - MEDIUM (6.7)

Description:

Any of the buy or sell functions of the contract allow creating a new **Order** even if the deposit amount is 0. This is allowed by the transfer function of the **Token** contract (called in the move_token function), since the check performed by check_nonnegative_amount function only verifies that the amount is less than 0, not equal.

Since these functions can be executed by any user, they could be used to stress the system through empty transactions, with the only handicap being the gas cost. This could lead eventually to a Denial of Service as the Counter key is incremented and there is no overflow handling, so at this point the code will **panic** every time it tries to create a new Order.

Code Location:

buy_limit function from **Cables** contract:

Listing 1: src/lib.rs (Lines 179,186,191,206)

```
164 pub fn buy_limit(
165     env: Env,
166     user: Address,
167     base_amnt: u128,
168     base_asst: Address,
169     quot_asst: Address,
170     max_price: u128,
171     exec_facility_alias: Symbol,
172 ) -> u128 {
173     // Authorize the `create` call by user to verify their
 ↳ identity.
174     user.require_auth();
175
176     let price_decim: u128 = 10_u128.pow(token_decimals(&env, &
 ↳ base_asst));
177
```

```
178     // deposit relevant token
179     let deposit: u128 = base_amnt * max_price / price_decim;
180     let deposit_token = quot_asst.clone();
181     move_token(
182         &env,
183         &deposit_token,
184         &user,
185         &env.current_contract_address(),
186         deposit.try_into().unwrap(),
187     );
188 
189     let new_order = Order {
190         creator: user.clone(),
191         balance: deposit,
192         quote_id: symbol_short!(""),
193         order_type: OrderType::Buy,
194         trade_type: TradeType::Limit,
195         base_asst: base_asst,
196         quot_asst: quot_asst,
197         base_amnt: base_amnt,
198         quot_amnt: 0,
199         limit_prc: max_price,
200         exec_fac: exec_facility_alias,
201         gas_dpst: 0,
202         slippage: 0,
203         status: OrderStatus::OpenUnfill,
204     };
205 
206     save_order(&env, new_order)
207 }
```

move_token function from **Cables** contract:

Listing 2: src/lib.rs (Line 1044)

```
1041 fn move_token(env: &Env, token: &Address, from: &Address, to: &
  ↳ Address, transfer_amount: i128) {
1042     // new token interface
1043     let token_client = token::Client::new(&env, &token);
1044     token_client.transfer(&from, to, &transfer_amount);
1045 }
```

transfer function from **Token** contract:

Listing 3: token/src/contract.rs (Line 110)

```
107 fn transfer(e: Env, from: Address, to: Address, amount: i128) {
108     from.require_auth();
109
110     check_nonnegative_amount(amount);
111
112     e.storage()
113         .instance()
114         .bump(INSTANCE_LIFETIME_THRESHOLD, INSTANCE_BUMP_AMOUNT);
115
116     spend_balance(&e, from.clone(), amount);
117     receive_balance(&e, to.clone(), amount);
118     TokenUtils::new(&e).events().transfer(from, to, amount);
119 }
```

check_nonnegative_amount function from **Token** contract:

Listing 4: token/src/contract.rs (Line 14)

```
13 fn check_nonnegative_amount(amount: i128) {
14     if amount < 0 {
15         panic!("negative amount is not allowed: {}", amount)
16     }
17 }
```

Proof of Concept:

This proof of concept shows how a user can create a large number of Orders at zero cost (excluding gas).

Listing 5: PoC_Test

```
1 #[test]
2 fn test_empty_orders() {
3     let env = Env::default();
4     env.mock_all_auths();
5     env.budget().reset_unlimited();
6
```

```
    mod wasm_contract {
        soroban_sdk::contractimport!(
            file = "./target/wasm32-unknown-unknown/release/
↳ soroban_cable_contract.wasm"
        );
    }
    let contract_id = &env.register_contract_wasm(None,
↳ wasm_contract::WASM);
    let client = CablesContractClient::new(&env, &contract_id);

    let u1 = Address::random(&env);
    let u2 = Address::random(&env);
    client.initialize(&u1, &0, &u1);

    let admin1 = Address::random(&env);
    let token_a = create_custom_token(&env, &admin1, &7);
    let token_b = create_custom_token(&env, &admin1, &7);

    // empty orders
    let number_of_orders = 100;
    for _ in 0..number_of_orders {
        client.buy_limit(
            &u2,
            &0,
            &token_a.address,
            &token_b.address,
            &0,
            &symbol_short!("ADMIN"),
        );
    }

    let orders = client.get_opened(&u2);

    println!("Number of orders: {:?}", orders.last().unwrap());
    assert_eq!(orders.last().unwrap(),99)

}
```

BVSS:

AO:A/AC:M/AX:L/C:N/I:N/A:C/D:N/Y:N/R:N/S:U (6.7)

Recommendation:

It is recommended to verify if the deposit amount to be transferred to the contract is 0, avoiding the creation of a new Order in this case.

Remediation Plan:

SOLVED: The Shift Markets team solved this issue in commit ID 96aa979 by introducing a verification of the transferred amount.

3.2 (HAL-02) LACK OF OVERFLOW CONTROL - MEDIUM (6.7)

Description:

The save_order function is responsible for storing in the ledger the information of a newly created Order and some tracking data related to it. One of this data is the CurOrder counter, which is incremented with each new **Order** created.

This counter is incremented without any overflow control, so in case the hypothetical maximum is reached, the operation will panic, and no new Order can be created.

It is recommended to handle counters with overflow controlled operations, such as checked_add, which returns a value None if the maximum is reached. A simple check of the value returned from the increment and a counter reset in case of None, would be sufficient to handle the situation.

Code Location:

Snippet of save_order function from **Cables** contract:

Listing 6: src/lib.rs (Line 864)

```
845     let order_hist: Vec<OrderChange> = vec![&env, order_change];
846     env.storage()
847         .persistent()
848         .set(&DataKey::OrderHist(cur_order), &order_hist);
849     env.storage().persistent().bump(
850         &DataKey::OrderHist(cur_order),
851         WEEK_LIFETIME_THRESHOLD,
852         WEEK_BUMP_AMOUNT,
853     );
854
855     // Publish an event about the new order saving.
856     // The event data is the id of the saved order.
857     let topics = (ORDER, order.creator, order.order_type, order.
 ↳ trade_type);
```

```
858     env.events().publish(
859         topics,
860         (cur_order, order.base_asst, order.quot_asst, order.status
 ↳ ),
861     );
862
863     // updating current order counter
864     cur_order += 1;
865     env.storage()
866         .persistent()
867         .set(&DataKey::CurOrder, &cur_order);
868     env.storage().persistent().bump(
869         &DataKey::CurOrder,
870         MONTH_LIFETIME_THRESHOLD,
871         MONTH_BUMP_AMOUNT,
872     );
873     // returns saved new order id
874     cur_order - 1
875 }
```

Proof of Concept:

Important: To perform this proof of concept, a new function has been introduced in the contract to set the CurOrder counter value close to the maximum in order to recreate the overflow more easily.

Listing 7: src/lib.rs

```
1 pub fn set_cur_order(env: Env, cur_order: u128) {
2     env.storage()
3         .persistent()
4         .set(&DataKey::CurOrder, &cur_order);
5     env.storage().persistent().bump(
6         &DataKey::CurOrder,
7         MONTH_LIFETIME_THRESHOLD,
8         MONTH_BUMP_AMOUNT,
9     );
10 }
```

This proof of concept shows the reaction of the contract to a CurOrder counter overflow.

Listing 8: PoC_Test

```rust
#[test]
#[should_panic] // Overflow
fn test_overflow() {
    let env = Env::default();
    env.mock_all_auths();
    env.budget().reset_unlimited();
    // let contract_id = env.register_contract(None, CablesContract);
    mod wasm_contract {
        soroban_sdk::contractimport!(
            file = "./target/wasm32-unknown-unknown/release/soroban_cable_contract.wasm"
        );
    }
    let contract_id = &env.register_contract_wasm(None, wasm_contract::WASM);
    let client = CablesContractClient::new(&env, &contract_id);

    let u1 = Address::random(&env);
    let u2 = Address::random(&env);
    client.initialize(&u1, &0, &u1);

    let admin1 = Address::random(&env);
    let token_a = create_custom_token(&env, &admin1, &7);
    let token_b = create_custom_token(&env, &admin1, &7);

    let max = u128::MAX - 1;

    client.set_cur_order(&max);

    client.buy_limit(
        &u2,
        &0,
        &token_a.address,
        &token_b.address,
        &0,
        &symbol_short!("ADMIN"),
    );

    println!("Everything OK at this point. Let's go to overflow panic!");

    client.buy_limit(
```

```
40          &u2,
41          &0,
42          &token_a.address,
43          &token_b.address,
44          &0,
45          &symbol_short!("ADMIN"),
46      );
47
48  }
```

BVSS:

AO:A/AC:M/AX:L/C:N/I:N/A:C/D:N/Y:N/R:N/S:U (6.7)

Recommendation:

It is recommended to handle counters with overflow controlled operations, such as checked_add, which returns a value None if the maximum is reached.

Remediation Plan:

SOLVED: The Shift Markets team solved this issue in commit ID f659149 by using the checked_add operation and handling the overflow error.

3.3 (HAL-03) OWNERSHIP CAN BE TRANSFERRED WITHOUT CONFIRMATION - LOW (2.5)

Description:

The set_admin function allows changing the owner/administrator of the contract. This function can only be executed by the current administrator, so if any unintentional error occurs in the execution of the transaction (e.g. an incorrect address), the control over the contract and privileged functionalities would be lost.

It is recommended to perform this type of operation using a two-step process: one transaction to set a value of the new_administrator and another transaction to confirm this information signed by the new administrator himself.

Code Location:

set_admin function from **Cables** contract:

Listing 9: src/lib.rs (Line 83)

```
80 pub fn set_admin(e: Env, new_admin: Address) {
81     let admin = read_administrator(&e);
82     admin.require_auth();
83     write_administrator(&e, &new_admin);
84
85     // Publish a set_admin event.
86     // The event data is address of new admin.
87     let topics = (SET_ADMIN,);
88     e.events().publish(topics, new_admin);
89 }
```

BVSS:

AO:A/AC:L/AX:H/C:N/I:N/A:H/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

It is recommended to split ownership transfer functionality into `set_owner` and `accept_ownership` functions. The latter function allows the transfer to be completed by the recipient.

Remediation Plan:

SOLVED: The Shift Markets team solved this issue in commit ID `1b6ee05` by introducing a two-step ownership transfer process.

3.4 (HAL-04) CONTRACT ADMINISTRATOR NOT ALLOWED TO FILL SOME ORDERS - INFORMATIONAL (1.7)

Description:

The fill function is responsible for matching buy/sell orders to perform the exchange of assets. The access control only allows execution to administrators (of the contract and execution facilities); however, the contract administrator will not be able to fill all orders.

At the start of the function, the alias of the execution facility linked to the executor's address is loaded, and subsequently it is checked that both **Orders** also have this alias. For this reason, the contract administrator will only be able to fill those orders that are executed in the execution facility ADMIN.

If this privilege is to be maintained for the contract administrator, the checks described above must include an exception for the contract administrator. Alternatively, access control for this role can be removed, leaving only the Self::is_execution_facility(env.clone(), executor.clone()) check.

Code Location:

Snippet of fill function from **Cables** contract:

Listing 10: src/lib.rs (Lines 703,710,711,748-750)

```
695 pub fn fill(
696     env: Env,
697     order_id1: u128,
698     order_id2: u128,
699     executor: Address,
700 ) -> Result<(), Error> {
701     let admin = read_administrator(&env);
702     // lock to the ExecFacility or admin
```

```
703     if Self::is_execution_facility(env.clone(), executor.clone())
 ↳ || executor == admin {
704         executor.require_auth();
705     } else {
706         panic!("not authorized")
707     }
708 
709     // load execution facility info
710     let key_alias = DataKey::ExecFacilityAlias(executor.clone());
711     let exec_alias: Symbol = env
712         .storage()
713         .persistent()
714         .get::<_, Symbol>(&key_alias)
715         .unwrap();
716     let exec_facility: ExecutionFacility =
717         Self::get_execution_facility(env.clone(), exec_alias.clone
 ↳ ());
718 
719     let cur_order: u128 = env
720         .storage()
721         .persistent()
722         .get(&DataKey::CurOrder)
723         .unwrap_or(0);
724     // require valid order id
725     if order_id1 > cur_order || order_id2 > cur_order {
726         panic_with_error!(&env, Error::UnknownOrder);
727     }
728     // load orders from storage
729     let mut maker: Order = env
730         .storage()
731         .persistent()
732         .get::<_, Order>(&DataKey::Order(order_id1))
733         .unwrap();
734     let mut taker: Order = env
735         .storage()
736         .persistent()
737         .get::<_, Order>(&DataKey::Order(order_id2))
738         .unwrap();
739     // Orders must be Sell and Buy, not same type
740     if maker.order_type == taker.order_type {
741         panic_with_error!(&env, Error::OrderTypesDoNotMatch);
742     }
743     // Orders must have the same Execution Facility
744     if maker.exec_fac != taker.exec_fac {
```

```
745             panic_with_error!(&env, Error::FacilitiesDoNotMatch);
746         }
747         // Orders must be filled by a specified Execution Facility
748         if maker.exec_fac != exec_alias {
749             panic_with_error!(&env, Error::FacilitiesDoNotMatch);
750         }
```

Proof of Concept:

This proof of concept shows the panic of the contract to a fill operation performed by the contract admin in an execution facility other than ADMIN.

Listing 11: PoC_Test

```
#[test]
#[should_panic(expected = "HostError: Error(Contract, #6)")] //
↳ FacilitiesDoNotMatch
fn test_fill_error_not_admin_allowed() {
    let env = Env::default();
    env.mock_all_auths();
    // env.budget().reset_unlimited();
    let contract_id = env.register_contract(None, CablesContract);
    let client = CablesContractClient::new(&env, &contract_id);

    let admin = Address::random(&env);
    let u1 = Address::random(&env);
    client.initialize(&admin, &1000, &admin);
    let exec1 = Address::random(&env);
    client.set_execution_facility(&symbol_short!("TEST1"), &exec1,
↳  &1000_u32, &exec1, &exec1);

    let (token_a, token_a_admin) = create_token(&env, &admin);
    let (token_b, token_b_admin) = create_token(&env, &admin);
    token_a_admin.mint(&u1, &200000);
    token_b_admin.mint(&u1, &220000);

    let sell_order1 = client.buy_limit(
        &u1,
        &100000,
        &token_a.address,
        &token_b.address,
        &(12 * 10_u128.pow(6_u32)),
```

```
27          &symbol_short!("TEST1"),
28      );
29      let sell_order2 = client.sell_limit(
30          &u1,
31          &100000,
32          &token_a.address,
33          &token_b.address,
34          &(10 * 10_u128.pow(6_u32)),
35          &symbol_short!("TEST1"),
36      );
37
38      client.fill(&sell_order1, &sell_order2, &admin);
39  }
```

BVSS:

AO:A/AC:L/AX:M/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (1.7)

Recommendation:

It is recommended to remove the access control for contract administrator from the `fill` function. Alternatively, if retaining this privilege is desired, the checks within the execution facility should be modified.

Remediation Plan:

SOLVED: The Shift Markets team solved this issue in commit ID `9dc6f23` by removing the access control for the contract administrator.

It has been clarified that there was no intent for administrator privileges, so the finding classification has been reduced from **Low** to **Info** as the only issue was that no specific access control was required for the contract administrator.

3.5 (HAL-05) ERROR HANDLING - INFORMATIONAL (1.7)

Description:

Throughout the contract, some errors are handled with the panic! macro, returning only a string message, and at other times the Error enum type is used to include the message in the panic_with_error! macro.

The fill and exchange functions are the only ones that return a Result type, but neither of them handle errors correctly by returning an Err object, instead they do so using the panic_with_error! macro, which causes the code to panic and stop execution.

In production environments, as a good practice for code hygiene and better debugging, it is recommended to use the Error type to handle errors, returning a Result(Ok, Error).

The use of this structure provides the advantage of error propagation via the ? symbol, thus avoiding the excessive use of the panic! macro.

Code Location:

initialize function from **Cables** contract:

Listing 12: src/lib.rs (Line 67)

```
65 pub fn initialize(e: Env, admin: Address, fee: u32, fee_receiver:
↳ Address) {
66     if has_administrator(&e) {
67         panic!("already initialized")
68     }
69     write_administrator(&e, &admin);
70     set_execution_facility(
71         e.clone(),
72         symbol_short!("ADMIN"),
73         admin.clone(),
74         fee,
75         fee_receiver,
```

```
76          admin.clone(),
77      );
78  }
```

cancel_order function from **Cables** contract:

Listing 13: src/lib.rs (Lines 774,783)

```
772         // require balance > 0
773         if order.balance == 0 {
774             panic!("zero balance");
775         }
776
777         let user = order.creator.clone();
778         let admin = read_administrator(&env);
779         // lock to the order creator or admin
780         if executor == user || executor == admin {
781             executor.require_auth();
782         } else {
783             panic!("not authorized")
784         }
785
786         cancel(env.clone(), order_id);
787     }
788 }
```

BVSS:

AO:A/AC:L/AX:M/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (1.7)

Recommendation:

It is recommended to use the Error type to handle errors, returning a Result(Ok, Error) from functions or using the macro panic_with_error!.

Remediation Plan:

SOLVED: The Shift Markets team solved this issue in commit ID 31f18c4 by improving the error handling using the Result type.

3.6 (HAL-06) LACK OF DEBUGGING INFO - INFORMATIONAL (1.7)

Description:

Most of the public entry points of the **Cables** contract do not return any information after execution.

As a good practice in production environments, for code hygiene and better debugging, it is recommended to incorporate event information into the main public functions to improve traceability and information logging.

As an example of this, many of the public entry points of the **Token** contract return transaction information using this `TokenUtils::new(&e).events()` functionality. It would be advisable to apply the same technique in the **Cables** contract.

Code Location:

`transfer` function from **Token** contract which returns event information:

Listing 14: token/src/contract.rs (Line 118)

```
107 fn transfer(e: Env, from: Address, to: Address, amount: i128) {
108     from.require_auth();
109
110     check_nonnegative_amount(amount);
111
112     e.storage()
113         .instance()
114         .bump(INSTANCE_LIFETIME_THRESHOLD, INSTANCE_BUMP_AMOUNT);
115
116     spend_balance(&e, from.clone(), amount);
117     receive_balance(&e, to.clone(), amount);
118     TokenUtils::new(&e).events().transfer(from, to, amount);
119 }
```

BVSS:

AO:A/AC:L/AX:M/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (1.7)

Recommendation:

It is recommended to return transaction information using this `TokenUtils ::new(&e).events()` functionality to increase traceability and debugging capabilities.

Remediation Plan:

SOLVED: The Shift Markets team solved this issue in commit ID `e48ad32` by adding new events to cover facilities and admin changes.

3.7 (HAL-07) MISUSE OF RUST HELPER METHODS - INFORMATIONAL (0.0)

Description:

The use of the unwrap or expect functions can be useful for testing purposes, as an error (panic!) is raised if the Option does not have a “Some” or “Result” values. Nevertheless, leaving unwrap or expect functions in production environments is considered bad practice as it will not only cause the program to crash out (panic!) but also, in the case of unwrap, no meaningful messages would be shown to help the user solve or understand the root cause of the error.

For example, if the reactivate_execution_facility function is called with an unknown or mistake alias, the code will panic with no error information.

Code Location:

Snippet of reactivate_execution_facility function from **Cables** contract:

Listing 15: src/exec.rs (Line 131)

```
112 pub fn reactivate_execution_facility(e: Env, alias: Symbol) {
113     let admin = read_administrator(&e);
114     admin.require_auth();
115
116     // load list of active facilities
117     let mut active_facilities: Vec<Symbol> = e
118         .storage()
119         .persistent()
120         .get(&DataKey::ActiveFacilities)
121         .unwrap_or(Vec::<Symbol>::new(&e));
122
123     if active_facilities.contains(&alias) {
124         panic!("Alias is already active")
125     }
126
127     active_facilities.push_back(alias.clone());
128
```

```
129     // loading saved facility
130     let key1 = DataKey::ExecutionFacility(alias);
131     let exec_facility: ExecutionFacility = e.storage().persistent
 ↳ ().get(&key1).unwrap();
132     // getting its address
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to avoid the use of unwrap or expect functions in production environments as it could cause a panic!, crashing the contract without error messages. Some alternatives are possible, such as propagating the error by putting a "?", using unwrap_or / unwrap_or_else / unwrap_or_default functions, or using error-chain crate for errors.

Reference: https://crates.io/crates/error-chain

Remediation Plan:

SOLVED: The Shift Markets team solved this issue in commit ID 31f18c4.

3.8 (HAL-08) ASSIGNMENT NOT NEEDED - INFORMATIONAL (0.0)

Description:

The exchange function makes some assignments to variables that are not necessary, since these values are previously checked by the fill function, which is the caller one.

The fill entry point performs the following checks:

- Both Orders have the same execution facility.
- The execution facility alias introduced as input is the same as the one in the Order.

Considering the above, the assignment of the execution alias that occurs at the beginning of the exchange function does not make sense.

Code Location:

initialize function from **Cables** contract:

Listing 16: src/lib.rs (Lines 608,610,611)

```
599 fn exchange(
600         env: &Env,
601         mut buy_order: Order,
602         mut sell_order: Order,
603         maker_type: OrderType,
604         execution_facility: ExecutionFacility,
605     ) -> Result<(u128, u128, Order, Order), Error> {
606         let fee_portion: u32 = execution_facility.fee;
607         let fee_receiver: Address = execution_facility.
 ↳ fee_receiver;
608         let exec_alias: Symbol = execution_facility.alias;
609 
610         buy_order.exec_fac = exec_alias.clone();
611         sell_order.exec_fac = exec_alias.clone();
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation:

It is recommended to remove useless code to save some gas.

Remediation Plan:

SOLVED: The Shift Markets team solved this issue in commit ID 050181f by removing the redundant code.

3.9 (HAL-09) UPDATE CONTRACT AND FACILITY ADMINISTRATOR AT SAME TIME - INFORMATIONAL (0.0)

Description:

The set_admin function allows modifying the contract administrator, which is created and stored during the initialize function. In the latter function, the ADMIN execution facility is also created, and its administrator is the same as the contract.

Since the set_admin function only modifies the contract administrator, it is recommended to add a call to set_facility_admin inside it to be able to also modify the ADMIN facility admin at the same time, avoiding that the facility remains with the old administrator address.

Code Location:

set_admin function from **Cables** contract:

Listing 17: src/lib.rs

```
80 pub fn set_admin(e: Env, new_admin: Address) {
81     let admin = read_administrator(&e);
82     admin.require_auth();
83     write_administrator(&e, &new_admin);
84
85     // Publish a set_admin event.
86     // The event data is address of new admin.
87     let topics = (SET_ADMIN,);
88     e.events().publish(topics, new_admin);
89 }
90
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to add a call to `set_facility_admin` to modify the administrator account of the contract and the `ADMIN` facility at the same time.

Remediation Plan:

SOLVED: The Shift Markets team solved this issue in commit ID `602f0bd` by adding a call to `set_facility_admin`, updating the facility administrator at the same time as the contract owner.

AUTOMATED TESTING

4.1 AUTOMATED ANALYSIS

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and vulnerabilities. Among the tools used was cargo audit, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in https://crates.io are stored in a repository named The RustSec Advisory Database. cargo audit is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. To better assist the developers maintaining this code, the auditors are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

Listing 18: Dependency tree

```
1 No security detections.
```

THANK YOU FOR CHOOSING

// HALBORN