SECURITY AUDIT OF

PROTON SALE SMART CONTRACT

Public Report

Nov 22, 2022

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Move	Move is a new programming language that implements all the transactions on the Aptos blockchain.
Move Module	A Move module defines the rules for updating the global state of the Aptos blockchain. In the Aptos protocol, a Move module is a smart contract.
Resource Account	A resource account is a developer feature used to manage resources independent of an account managed by a user, specifically publishing modules and automatically signing for transactions
LP tokens	Liquidity Provider tokens are tokens issued to liquidity providers on a decentralized exchange (DEX) that run on an automated market maker (AMM) protocol. Uniswap, Sushi and PancakeSwap are some examples of popular DEXs that distribute LP tokens to their liquidity providers.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Nov 22, 2022. We would like to thank the Proton Sale for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Proton Sale Smart Contract. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Proton Sale Smart Contract

Proton Sale Smart Contract is a launchpad platform that is specifically designed for launching new coins, crypto projects, and raising liquidity.

Launchpads help investors discover early-stage crypto projects before they enter into the mainstream. Over the years, launchpads have relied upon human verification system to determine which projects or tokens to launch. However, with the advancement of the blockchain and smart contracts, new launchpads are now choosing a different and more decentralized approach by building trustless systems through smart contracts that work together to enhance the experience of token management for businesses.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of Proton Sale Smart Contract. It was conducted on commit `263d081d99119bf85867eb3b7d085a6584bd3789` from git repository link: *https://github.com/0xmodule/proton-sale*.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
`0c363c0686c53a8a4d04bdd422b18b7452e6380a52787c08df27e0653c36e342`	`aggregator.move`
`ac5401b21e21e4dd25aec0231e1db1eebc1169a6a57011394e2365ed19f0e293`	`global_state.move`
`d31a516fe5c826e47b70fc5289b977682166f359075d19e1cd27694953853d41`	`pausable.move`
`c1564ca876d267e9c9a0590abe27d6f14972c1bfa6db0d5cb1f8a2ad6fec20ae`	`sale.move`
`d603d3d65d32a9f70699126ead61a646f4633eb33ac0ba1745cb4fbded0a83c0`	`to_string.move`

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Numerical precision errors
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- Gas Usage, Gas Limit and Loops
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The Proton Sale Smart Contract was written in `Move` programming language. It's a launchpad platform where users can launch new crypto projects and raise liquidity. The processing is divided into:

2.1.1. Create a sale

Crypto project owners create their sale, deposit sales tokens, register required information such as purchase tokens and rate, soft cap, hard-cap, sale duration, liquidity providing, LP lock duration, vesting schedule, ... and paying fee.

2.1.2. Pre-start

The owner can add users to whitelist in this period.

2.1.3. Contribution period

Users can now see the project information and contribute to the project (buy tokens using purchase tokens). There are 2 phases in this period: the first one is only for whitelisted users and the next one for everyone but the sale will stop when the total contribution amount reach the hard cap.

2.1.4. Sale end period

The sale is success if the sale reaches the soft cap (total amount of contribution are larger than the soft cap amount). Otherwise, investors contributed to the project can get a full refund for their contribution and project owner can withdraw all sale tokens back.

2.1.5. Finalization

If the sale is success, the project owner can now finalize the sale to add liquidity (using either PancakeSwap or Liquidswap), lock the LP tokens and create vesting schedule for all investors. LP tokens locked can be claimed by the project owner after locking duration registered in the first step is over. Leftover sale tokens (which have not been sold) still remained in the contract.

2.1.6. TGE + Vesting

Investors can now claim their TGE tokens and continue to claim remaining tokens follow by vesting schedule created.

2.1.7. Emergency state

Proton Sale Admin can pause the contract in case of emergency. When the contract is paused, all the functionality is temporary disable until resuming. Admin can withdraw all tokens (including locked liquidity) out of the contract in this state. Admin can also disable these emergency functions forever.

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of Proton Sale Smart Contract.

Proton Sale fixed the code, according to Verichains's private reports, in commit `56d660f83fa6a34596a4fdbd1d508088d499c4bc`.

2.2.1. sale.move - Unable to claim tokens with zero `tge_percent` HIGH

When `create_sale`, if users create a sale with 0 percent TGE (launch project without TGE tokens release, only vesting). The investors are then unable to claim their vesting tokens because `tge_token` is zero which make `claimed_amount` become zero.

```
public entry fun claim<SaleCoinType, PurchaseCoinType>(
    account: &signer,
    creator: address,
    id: vector<u8>,
    minter_addr: address,
    counter: u64,
) acquires SaleStore, ProtonEvent {
    ...
    let hardcap = table::borrow(&creator_sale_store.sale_infos, sale_id).hardcap;
    let sale_coin_amount = table::borrow(&creator_sale_store.sale_infos,
sale_id).sale_coin_amount;
    let tge_percent = table::borrow(&creator_sale_store.vesting_infos,
sale_id).tge_percent;
    let tge_time = table::borrow(&creator_sale_store.vesting_infos, sale_id).tge_time;
    let purchased = table::borrow(&nft_store.purchase_tickets, sale_id).purchased_amount;
    let claimed_amount = property_map::read_u64(&property_map, &string::utf8(CLAIMED));
    let allocation = property_map::read_u64(&property_map, &string::utf8(ALLOCATION));
    if (!is_account_registered<SaleCoinType>(account_addr)) {
        register<SaleCoinType>(account);
    };

    if (purchased > 0 && claimed_amount == 0) {
        let tge_token = (((((purchased as u128) * (sale_coin_amount as u128)) / (hardcap as
u128) ) * (tge_percent as u128) / 100) as u64) + 0;
        coin::transfer<SaleCoinType>(&resource_signer, account_addr, tge_token);
        mutate_tokendata_property(
            &resource_signer,
            token_data_id,
            vector<String>[string::utf8(CLAIMED)],
            vector<vector<u8>>[ bcs::to_bytes<u64>(&tge_token)],
            vector<String>[ string::utf8(b"u64")],
        );

        let process = ((((tge_token as u128) * 10) / (allocation as u128)) as u8) + 48;

        let creator_addr_str =
to_string::bytes_to_hex_string(&bcs::to_bytes<address>(&creator));

        let uri = b"https://s3.ap-southeast-1.amazonaws.com/module-
labs.proton.file/images/";
        vector::append(&mut uri, bcs::to_bytes<String>(&creator_addr_str));
        vector::append(&mut uri, b"/");
        vector::append(&mut uri, bcs::to_bytes<u8>(&process));
        vector::append(&mut uri, b".png");

        mutate_tokendata_uri(&resource_signer, token_data_id, string::utf8(uri));
    };
    claimed_amount = property_map::read_u64(&property_map, &string::utf8(CLAIMED));
    if (purchased > 0 && claimed_amount > 0) {
        ...
    };
}
```

RECOMMENDATION

If the contract does not support zero percent TGE tokens release, add an assert statement into creating sale step. Otherwise, let the investors claim vesting when TGE percent is zero instead of depending on `claimed_amount > 0`.

UPDATES

- *Nov 16, 2022*: This issue has been acknowledged and fixed by the Proton Sale team.

2.2.2. pausable.move - Wrong account for `disable_forever` HIGH

`disable_forever` function use wrong account to get `emergency_account_cap` so emergency admin can not `disable_forever` pausable module.

```
public entry fun disable_forever(account: &signer) acquires EmergencyAccountCapability {
    assert!(!is_disabled(), ERR_DISABLED);
    assert!(signer::address_of(account) == global_state::get_emergency_admin(),
ERR_NO_PERMISSIONS);

    let emergency_account_cap =
    borrow_global<EmergencyAccountCapability>(@liquidswap);
    let emergency_account =
account::create_signer_with_capability(&emergency_account_cap.signer_cap);
```

```
    move_to(&emergency_account, IsDisabled {});
}
```

RECOMMENDATION

Replacing `liquidswap` account with `proton_sale` account.

```
public entry fun disable_forever(account: &signer) acquires EmergencyAccountCapability {
    assert!(!is_disabled(), ERR_DISABLED);
    assert!(signer::address_of(account) == global_state::get_emergency_admin(),
ERR_NO_PERMISSIONS);

    let emergency_account_cap =
    borrow_global<EmergencyAccountCapability>(@proton_sale);
    let emergency_account =
account::create_signer_with_capability(&emergency_account_cap.signer_cap);
    move_to(&emergency_account, IsDisabled {});
}
```

UPDATES

- *Nov 15, 2022*: This issue has been acknowledged and fixed by the Proton Sale team.

2.2.3. sale.move - Cannot claim when `counter` is larger than 9999 MEDIUM

When there are more than 9999 contributors, the next ones can purchase but unable to `mint_nft` and `claim` tokens because the expression `i < 4 - string::length(&counter_string)` will make these functions reverted (mathematics overflow).

```
public entry fun mint_nft<SaleCoinType, PurchaseCoinType>(
    account: &signer,
    creator: address,
    id: vector<u8>,
) acquires SaleStore {
    ...
    let counter = &mut table::borrow_mut(&mut creator_sale_store.nft_counter,
sale_id).count;
    let seed = &mut to_string::bytes_to_hex_string(&bcs::to_bytes<address>(&account_addr));
    let counter_string = to_string::to_string((*counter as u128) + 0);
    string::append(seed, string::utf8(b" #"));
    let i = 0;
    while (i < 4 - string::length(&counter_string)) {
        string::append(seed, string::utf8(b"0"));
        i = i + 1;
    };
    string::append(seed, counter_string);
    ...
}

public entry fun claim<SaleCoinType, PurchaseCoinType>(
    account: &signer,
    creator: address,
    id: vector<u8>,
    contributor: address,
    counter: u64,
) acquires SaleStore, ProtonEvent {
    ...
    let counter_string = to_string::to_string((counter as u128) + 0);
    string::append(seed, string::utf8(b" #"));
    let i = 0;
    while (i < 4 - string::length(&counter_string)) {
        string::append(seed, string::utf8(b"0"));
        i = i + 1;
    };
    string::append(seed, counter_string);
    ...
}
```

RECOMMENDATION

Adding padding properly to `counter_string` instead of hardcoded 4 or limit number of contributor < 10000. The padding logic should be put into a seperated function to avoid logic duplication.

UPDATES

- *Nov 22, 2022*: This issue has been acknowledged and fixed by the Proton Sale team.

2.2.4. sale.move - Invalid `amount` in `get_vesting_claimable` MEDIUM

The condition for returning no tokens claimable must be `tge_percent == 100` instead of `vesting_cycle_percent == 100`. Current condition will make vesting tokens unclaimable if the `vesting_cycle_percent` is set to 100.

```
public fun get_vesting_claimable<SaleCoinType, PurchaseCoinType>(
    hardcap: u64,
    sale_coin_amount: u64,
    tge_percent: u8,
    tge_time: u64,
    vesting_cycle_duration: u64,
    vesting_cycle_percent: u8,
    claimed_amount: u64,
    purchased: u64,
): u64 {
    if (timestamp::now_seconds() < (tge_time + vesting_cycle_duration) ||
vesting_cycle_percent == 100 || tge_time == 0) {
        return 0
    };
    let token_can_claim = (purchased * sale_coin_amount) / hardcap;
    let non_vesting = (token_can_claim * (tge_percent as u64)) / 100;
    let for_vesting = token_can_claim - non_vesting;
    let time_pass = timestamp::now_seconds() - tge_time;
    let vesting_claimed = claimed_amount;
    if (claimed_amount > 0) {
        vesting_claimed = vesting_claimed - non_vesting;
    };
    let token_unlock = ((time_pass / vesting_cycle_duration) * (vesting_cycle_percent as
u64) * token_can_claim) / 100;
    if (token_unlock > for_vesting) {
        return for_vesting - vesting_claimed
    } else {
        return token_unlock - vesting_claimed
    }
}
```

RECOMMENDATION

Replacing `vesting_cycle_percent == 100` with `tge_percent == 100`. We also need to validate all percentage condition carefully when creating a sale to avoid unexpected result.

```
public fun get_vesting_claimable<SaleCoinType, PurchaseCoinType>(
    hardcap: u64,
    sale_coin_amount: u64,
    tge_percent: u8,
    tge_time: u64,
    vesting_cycle_duration: u64,
    vesting_cycle_percent: u8,
    claimed_amount: u64,
    purchased: u64,
): u64 {
    if (timestamp::now_seconds() < (tge_time + vesting_cycle_duration) || tge_percent ==
100 || tge_time == 0) {
        return 0
    };
    let token_can_claim = (purchased * sale_coin_amount) / hardcap;
    let non_vesting = (token_can_claim * (tge_percent as u64)) / 100;
    let for_vesting = token_can_claim - non_vesting;
    let time_pass = timestamp::now_seconds() - tge_time;
    let vesting_claimed = claimed_amount;
    if (claimed_amount > 0) {
        vesting_claimed = vesting_claimed - non_vesting;
    };
    let token_unlock = ((time_pass / vesting_cycle_duration) * (vesting_cycle_percent as
u64) * token_can_claim) / 100;
    if (token_unlock > for_vesting) {
        return for_vesting - vesting_claimed
    } else {
        return token_unlock - vesting_claimed
    }
}
```

UPDATES

- *Nov 16, 2022*: This issue has been acknowledged and fixed by the Proton Sale team.

2.2.5. sale.move - Cannot claim when `purchased` is small LOW

In `claim` function, when a user `contribute` with a small amount that `(purchased * sale_coin_amount / hardcap) * tge_percent` is less than 100, `tge_token` will be zero so the user can claim neither TGE nor vesting tokens.

```
public entry fun claim<SaleCoinType, PurchaseCoinType>(
    account: &signer,
    creator: address,
    id: vector<u8>,
    minter_addr: address,
) acquires SaleStore, ProtonEvent {
    ...
    let hardcap = table::borrow(&creator_sale_store.sale_infos, sale_id).hardcap;
    let sale_coin_amount = table::borrow(&creator_sale_store.sale_infos,
sale_id).sale_coin_amount;
    let tge_percent = table::borrow(&creator_sale_store.vesting_infos,
sale_id).tge_percent;
    let tge_time = table::borrow(&creator_sale_store.vesting_infos, sale_id).tge_time;
    let purchased = table::borrow(&nft_store.purchase_tickets, sale_id).purchased_amount;
    let claimed_amount = property_map::read_u64(&property_map, &string::utf8(b"Claimed"));
    let allocation = property_map::read_u64(&property_map, &string::utf8(b"Allocation"));
        if (!is_account_registered<SaleCoinType>(account_addr)) {
        register<SaleCoinType>(account);
    };
    if (purchased > 0 && claimed_amount == 0) {
        let tge_token = (((((purchased as u128) * (sale_coin_amount as u128)) / (hardcap as
u128) ) * (tge_percent as u128) / 100) as u64) + 0;
        coin::transfer<SaleCoinType>(&resource_signer, account_addr, tge_token);
        let amount = claimed_amount + tge_token;
        mutate_tokendata_property(
        &resource_signer,
        token_data_id,
        vector<String>[string::utf8(b"Claimed")],
        vector<vector<u8>>[ bcs::to_bytes<u64>(&amount)],
        vector<String>[ string::utf8(b"u64")],
        );

        let process = ((((amount as u128) * 10) / (allocation as u128)) as u8) + 48;
        let uri = b"https://files.protonsale.io/collections/";
        vector::append(&mut uri, id);
        vector::append(&mut uri, b"/");
        vector::append(&mut uri, bcs::to_bytes<u8>(&process));
        vector::append(&mut uri, b".png");
        mutate_tokendata_uri(&resource_signer, token_data_id, string::utf8(uri));
    };
    ...
}
```

RECOMMENDATION

Checking if the contribution amount is not so small that make `tge_token` become zero.

UPDATES

- *Nov 15, 2022*: This issue has been acknowledged by the Proton Sale team.

2.2.6. aggregator.move - Zero `amount_x_min` and `amount_y_min` could lead to unlimited slippage if the pool is already existed INFORMATIVE

Zero `amount_x_min` and `amount_y_min` could lead to unlimited slippage if the pool is already existed. Unlimited slippage could be used by attackers to drain the liquidity with sandwich attack.

```
public (friend) fun add_liquidity<SaleCoinType, PurchaseCoinType>(
    resource_signer: &signer,
    creator: &signer,
    dex_type: u8,
    sale_coin_amount: u64,
    purchase_coin_amount: u64
) : (TypeInfo, u64) {
    ...
    add_liquidity<SaleCoinType, PurchaseCoinType>(resource_signer, sale_coin_amount,
purchase_coin_amount, 0, 0); // unlimited slippage
    ...
}
```

RECOMMENDATION

Consider using `sale_coin_amount` and `purchase_coin_amount` for `amount_x_min` and `amount_y_min` or let users specify them.

UPDATES

- *Nov 15, 2022*: This issue has been acknowledged and fixed by the Proton Sale team.

2.2.7. sale.move - Duplicated contribute logic code in INFORMATIVE

In `contribute` function, there is a duplicated contribute code in 2 if conditions for whitelist and non whitelist.

```
public entry fun contribute<SaleCoinType, PurchaseCoinType>(
    account: &signer,
    creator: address,
    id: vector<u8>,
    amount: u64,
    random_seed: vector<u8>
) acquires SaleStore, ProtonEvent {
    ...
    if (whitelist_duration > 0 && timestamp::now_seconds() < sale_start_time +
whitelist_duration) {
        assert!(is_account_whitelisted<SaleCoinType, PurchaseCoinType>(account_addr,
creator, id), E_NOT_WHITELISTED);
        // duplicated contribute logic
    }
    if (timestamp::now_seconds() >= sale_start_time + whitelist_duration &&
timestamp::now_seconds() <= sale_start_time + whitelist_duration + non_whitelist_duration)
{
        // duplicated contribute logic
    }
    ...
}
```

RECOMMENDATION

Consider using the same contribute logic with condition checking. Also using `get_sale_period` function instead of calculate time period here.

```
public entry fun contribute<SaleCoinType, PurchaseCoinType>(
    account: &signer,
    creator: address,
    id: vector<u8>,
    amount: u64,
    random_seed: vector<u8>
) acquires SaleStore, ProtonEvent {
    ...
    let period: u8 = get_sale_period<SaleCoinType, PurchaseCoinType>(creator, id);

    assert!( period == NON_WHITELIST || period == WHITELIST, E_SALE_ENDED);

    assert!( period == NON_WHITELIST || is_account_whitelisted<SaleCoinType,
PurchaseCoinType>(account_addr, creator, id), E_NOT_WHITELISTED);

    // contributed logic code here
    ...
}
```

UPDATES

- *Nov 15, 2022*: This issue has been acknowledged and fixed by the Proton Sale team.

2.2.8. sale.move - Hardcode string variables INFORMATIVE

There are some hardcoded properties in many places in the contract. We should make constant for each property to avoid typo which could make wrong read/write while using these properties.

```
string::utf8(b"Minter"),
string::utf8(b"Claimed"),
string::utf8(b"Allocation"),
string::utf8(b"Cointype")
```

RECOMMENDATION

Adding constants for all properties instead of hardcoding.

UPDATES

- *Nov 15, 2022*: This issue has been acknowledged and fixed by the Proton Sale team.

2.2.9. sale.move - Duplicated assert INFORMATIVE

There is a duplicated assert in `claim` function. `claimed_amount` is zero in the if statement `purchased > 0 && claimed_amount == 0` so we don't need to calculate new `amount`.

```
public entry fun claim<SaleCoinType, PurchaseCoinType>(
    account: &signer,
    creator: address,
    id: vector<u8>,
    minter_addr: address,
) acquires SaleStore, ProtonEvent {
    ...
    assert!(
        exists<SaleStore<SaleCoinType, PurchaseCoinType>>(minter_addr),
        E_SALE_NOT_EXISTS,
    ); // duplicated assert
    assert!(table::contains(&nft_store.purchase_tickets, sale_id), E_NFT_NOT_EXISTS);
    let seed = table::borrow(&nft_store.purchase_tickets, sale_id).random_seed;

    let token_data_id = create_token_data_id(
    signer::address_of(&resource_signer),
    string::utf8(id),
    string::utf8(seed),
    );

    let token_id = create_token_id_raw(
        signer::address_of(&resource_signer),
        string::utf8(id),
        string::utf8(seed),
        0,
    );
    let balance = balance_of(account_addr, token_id);
    assert!(balance == 1, E_NOT_OWN_NFT);
    let property_map = get_property_map(account_addr, token_id);
    assert!(
        exists<SaleStore<SaleCoinType, PurchaseCoinType>>(minter_addr),
        E_SALE_NOT_EXISTS,
    ); // duplicated assert
    ...

    if (purchased > 0 && claimed_amount == 0) {
        let tge_token = (((((purchased as u128) * (sale_coin_amount as u128)) / (hardcap as
u128) ) * (tge_percent as u128) / 100) as u64) + 0;
        coin::transfer<SaleCoinType>(&resource_signer, account_addr, tge_token);
        let amount = claimed_amount + tge_token; // claimed_amount is zero, unnecessary
calculation.
        mutate_tokendata_property(
            &resource_signer,
            token_data_id,
            vector<String>[string::utf8(b"Claimed")],
            vector<vector<u8>>[ bcs::to_bytes<u64>(&amount)],
            vector<String>[ string::utf8(b"u64")],
        );

        let process = ((((amount as u128) * 10) / (allocation as u128)) as u8) + 48;
        let uri = b"https://files.protonsale.io/collections/";
        vector::append(&mut uri, id);
        vector::append(&mut uri, b"/");
        vector::append(&mut uri, bcs::to_bytes<u8>(&process));
        vector::append(&mut uri, b".png");
        mutate_tokendata_uri(&resource_signer, token_data_id, string::utf8(uri));
    };
    ...
}
```

RECOMMENDATION

Removing the duplication. Remove unnecessary `amount` calculation.

UPDATES

- *Nov 15, 2022*: This issue has been acknowledged and fixed by the Proton Sale team.

2.2.10. sale.move - High gas cost with large whitelist INFORMATIVE

The whitelist is currently using vector for storing. Please note that with the increase of the vector length overtime, the gas cost to perform operations like `contain`, `index_of` and `remove` is increasing more and more for every transaction using whitelist.

We can also bring the `list` variable out of the loop and use `swap_remove` instead of `remove` (we don't need the order here) in `remove_whitelist` to save some gas.

```
public entry fun add_whitelist<SaleCoinType, PurchaseCoinType>(
    account: &signer,
    id: vector<u8>,
    creator: address,
    addresses: vector<address>
) acquires SaleStore {
    let account_addr = signer::address_of(account);
    let sale_id = get_sale_id(creator, id);
    let creator_sale_store = borrow_global_mut<SaleStore<SaleCoinType,
PurchaseCoinType>>(creator);
    check_sale_store_exists<SaleCoinType, PurchaseCoinType>(account);
    assert!(
        table::contains(&creator_sale_store.whitelists, sale_id),
        E_SALE_NOT_EXISTS,
    );
    assert!(account_addr == creator, E_NOT_CREATOR);
    let i = 0;
    assert!(vector::length(&addresses) <= 5, E_TOO_MUCH_ADDRESS);
    while (i < vector::length(&addresses)) {
        let addr = *vector::borrow(&addresses, i);
        let list = &mut table::borrow_mut(&mut creator_sale_store.whitelists,
sale_id).list; // bring this out of the loop
        assert!(!vector::contains(list, &addr), E_CONTAIN_IN_LIST);
        vector::push_back(list, addr);
        i = i + 1;
    }
}

public entry fun remove_whitelist<SaleCoinType, PurchaseCoinType>(
    account: &signer,
    id: vector<u8>,
    creator: address,
    addresses: vector<address>
) acquires SaleStore {
    let account_addr = signer::address_of(account);
    let sale_id = get_sale_id(creator, id);
    let creator_sale_store = borrow_global_mut<SaleStore<SaleCoinType,
PurchaseCoinType>>(creator);
    assert!(
        table::contains(&creator_sale_store.sale_infos, sale_id),
        E_SALE_NOT_EXISTS,
    );
    assert!(account_addr == creator, E_NOT_CREATOR);
    let i = 0;
    assert!(vector::length(&addresses) <= 5, E_TOO_MUCH_ADDRESS);
    while (i < vector::length(&addresses)) {
        let addr = *vector::borrow(&addresses, i);
        let list = &mut table::borrow_mut(&mut creator_sale_store.whitelists,
sale_id).list; // bring this out of the loop
        let (found, index) = vector::index_of(list, &addr);
        assert!(found, E_CONTAIN_IN_LIST);
        vector::remove(list, index); // using swap_remove instead of remove
    }
}
```

UPDATES

- *Nov 16, 2022*: This issue has been acknowledged by the Proton Sale team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Nov 15, 2022*	Private Report	Verichains Lab
1.1	*Nov 16, 2022*	Private Report	Verichains Lab
1.2	*Nov 22, 2022*	Public Report	Verichains Lab

Table 2. Report versions history