Security Audit Report for Satoshi-Account

Date: December 18, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Satoshi
Target	Satoshi-Account

Version History

Version	Date	Description
1.0	December 18, 2024	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-02-19 04:13:39+0000

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Rust
Approach	Semi-automatic and manual verification

The repository that has been audited includes Satoshi-Account [1].

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Satoshi-Account	`Version 1`	`dd660a9eb5e8982e8b163928e787b8fd9e9830ab`
	`Version 2`	`481a7c106dad0b23aa3a4e0786efee12a358196d`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

[1] https://github.com/Near-Bridge-Lab/btc-bridge-contract/tree/v0.3.0/contracts/satoshi-account

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization

  * Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

		Likelihood	
		High	*Low*
Impact	*High*	High	Medium
	Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **seven** potential issues and **three** notes as follows:

- High Risk: 2
- Medium Risk: 2
- Low Risk: 3
- Note: 3

ID	Severity	Description	Category	Status
1	High	Manipulation of account's state by malicious relayer	DeFi Security	Fixed
2	Low	Incorrect check in function `withdraw_inner_account_yoctonear()`	DeFi Security	Fixed
3	Low	Bypassed debt settlement in function `request_sign_near_txs()` leading to gas and protocol fee loss	DeFi Security	Confirmed
4	Medium	Lack of updating global variables in function `csna_verify_deposit()`	DeFi Security	Fixed
5	High	Overwritten debt_info leading to incorrect user debt	DeFi Security	Fixed
6	Low	Lack of callback function to handle the return value of function `create_account()`	DeFi Security	Fixed
7	Medium	Lack of check on contract_id and deposited amount in function `handle_extra_msg()`	DeFi Security	Confirmed
8	-	Potential centralization risk	Note	-
9	-	Potential delayed price due to untimely price updated from oracles	Note	-
10	-	No guarantee of transaction success on NEAR and refund protocol_fee on failure	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Manipulation of account's state by malicious relayer

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `csna_verify_deposit()` is designed to assist `BTC` users in verifying deposits with extra actions. Before the `bridge` contract verifies the deposits, the state of the account (e.g., `yoctonear_amount`) is updated.

However, the cross contract invocation (i.e., `verify_deposit()`) may fail and the updated state will not be rolled back. In this case, malicious `relayers` can feed a fake deposit message to manipulate the account's state.

```
#[payable]
pub fn csna_verify_deposit(
    &mut self,
    deposit_msg: DepositMsg,
    tx_bytes: Vec<u8>,
    vout: usize,
    tx_block_blockhash: String,
    tx_index: u64,
    merkle_proof: Vec<String>,
) - > Promise {
    self.assert_contract_running();
    require!(
        env::prepaid_gas() == Gas::from_tgas(300),
        "Insufficient gas"
    );


    let extra_msg_used_amount = if let Some(extra_msg_str) = deposit_msg.extra_msg.as_ref() {
        self.handle_extra_msg(&deposit_msg.recipient_id, extra_msg_str)
    } else {
        0
    };


    let relayer_deposit = env::attached_deposit().as_yoctonear();
    let account = self.internal_unwrap_mut_account(&deposit_msg.recipient_id);
    if extra_msg_used_amount > account.yoctonear_amount {
        require!(
            extra_msg_used_amount == relayer_deposit,
            format!("Need deposit {} yoctonear", extra_msg_used_amount)
        );
        account.debt_info = Some(DebtInfo {
            near_gas_debt_amount: 0,
            protocol_fee_debt_amount: 0,
            relayer_debt_info: Some(RelayerDebtInfo {
                relayer_id: env::predecessor_account_id(),
                total_debt_amount: relayer_deposit,
                repay_debt_amount: 0,
            }),
        });
    } else {
        account.yoctonear_amount -= extra_msg_used_amount;
        if relayer_deposit > 0 {
            Promise::new(env::predecessor_account_id())
                .transfer(NearToken::from_yoctonear(relayer_deposit));
        }
    }
    ext_bridge::ext(self.internal_config().bridge_account_id.clone())
        .with_static_gas(GAS_FOR_BRIDGE_VERIFY_DEPOSIT)
        .verify_deposit(
            deposit_msg,
            tx_bytes,
            vout,
            tx_block_blockhash,
            tx_index,
            merkle_proof,
        )
}
```

Listing 2.1: bridge.rs

Impact Users' state can be manipulated.

Suggestion Implement a whitelist mechanism to restrict access to the function, allowing only whitelisted `relayers` to invoke it.

2.1.2 Incorrect check in function `withdraw_inner_account_yoctonear()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Users can invoke the function `withdraw_inner_account_yoctonear()` to withdraw their deposited `gas_token` in the form of `NEAR`. If the user specifies an empty `amount`, it indicates withdrawing the entire balance. However, during the check, it incorrectly requires the `NEAR` amount to be less than the total amount the user owns, rather than less than or equal to it. This means the user cannot withdraw the entire balance, which is incorrect.

The same issue also exists in the functions `withdraw_protocol_fee()` and `withdraw_collected_gas_token()`.

```
pub fn withdraw_inner_account_yoctonear(&mut self, amount: Option<U128>) {
    self.assert_contract_running();
    assert_one_yocto();
    let csna_id = env::predecessor_account_id();
    let account = self.internal_unwrap_mut_account(&csna_id);
    require!(
```

```rust
            account.debt_info.is_none(),
            "The account has outstanding debt."
        );
        let total_amount = account.yoctonear_amount;
        let amount = amount.map(|v| v.0).unwrap_or(total_amount);
        require!(amount > 0 && amount < total_amount, "Invalid amount");
        account.yoctonear_amount -= amount;
        require!(
            self.data().cur_available_near_gas >= amount,
            "Insufficient cur_available_near_gas"
        );
        self.data_mut().cur_available_near_gas -= amount;
        self.data_mut().acc_distributed_near_gas += amount;
        Event::WithdrawInnerAccountGasToken {
            csna_id: &csna_id,
            amount: amount.into(),
        }
        .emit();
        Promise::new(csna_id).transfer(NearToken::from_yoctonear(amount));
    }
```

Listing 2.2: csna.rs

```rust
    pub fn withdraw_protocol_fee(&mut self, amount: Option<U128>) {
        self.assert_contract_running();
        assert_one_yocto();
        self.assert_owner();
        let total_amount = self.data_mut().cur_available_protocol_fee;
        let amount = amount.map(|v| v.0).unwrap_or(total_amount);
        require!(amount > 0 && amount < total_amount, "Invalid amount");
        self.data_mut().cur_available_protocol_fee -= amount;
        self.data_mut().acc_claimed_protocol_fee += amount;
        Promise::new(self.internal_config().owner_id.clone())
            .transfer(NearToken::from_yoctonear(amount));
        Event::WithdrawProtocolFee {
            account_id: &env::predecessor_account_id(),
            amount: amount.into(),
        }
        .emit();
    }
```

Listing 2.3: management.rs

```rust
    pub fn withdraw_collected_gas_token(
        &mut self,
        token_id: AccountId,
        amount: Option<U128>,
    ) -> Promise {
        self.assert_contract_running();
        assert_one_yocto();
        self.assert_owner();
        let total_amount = self
            .data_mut()
            .collected_gas_token
            .remove(&token_id)
            .expect("Invalid token_id");
        let amount = amount.map(|v| v.0).unwrap_or(total_amount);
        require!(amount > 0 && amount < total_amount, "Invalid amount");
        let remain = total_amount - amount;
        if remain > 0 {
            self.data_mut()
                .collected_gas_token
                .insert(token_id.clone(), remain);
        }
        self.transfer_collected_gas_token(&token_id, amount)
    }
```

Listing 2.4: management.rs

Impact Users are unable to withdraw their entire balance due to the incorrect check on the NEAR amount, leading to potential withdrawal issues.

Suggestion Revise the logic to allow the withdrawal amount to be equal to total_amount.

2.1.3 Bypassed debt settlement in function request_sign_near_txs() leading to gas and protocol fee loss

Severity Low

Status Confirmed

Introduced by Version 1

Description The function request_sign_near_txs() allows users to generate the necessary information for chain signing even when the NEAR balance in their internal account is insufficient to cover the gas fee and protocol fee. However, it requires that the first transaction in the signing request involves the user depositing the total required fee into their internal account to repay the debt.

However, the function does not ensure the user's account has sufficient gas tokens. Even if the first transaction fails, the subsequent transactions can still be executed. This bypasses the debt settlement step, resulting in the contract failing to collect the protocol fee and gas fee.

```
    pub fn request_sign_near_txs(&mut self, user_intention: UserIntention) {
        self.assert_contract_running();
        let near_transactions_bytes = user_intention.get_near_transactions_bytes();
        let near_pending_id = generate_near_pending_sign_id(&near_transactions_bytes);
        require!(
            !self
                .data()
                .near_pending_sign_txs
                .contains_key(&near_pending_id),
            "Repeated user intention"
        );
        user_intention.verify_signature();
        require!(
            user_intention.intention.chain_id == NEAR_CHAIN_ID,
            "Unsupported chain id"
        );
        let csna_path = get_csna_path(&user_intention.btc_public_key);
        let csna_id = self.get_csna_id(&csna_path);
        require!(
            csna_id == user_intention.intention.csna,
            "csna does not match btc public key"
        );
        let account = self.internal_unwrap_account(&csna_id);
        require!(
            account.nonce == user_intention.intention.nonce,
            "Invalid nonce"
        );
        require!(
            account.near_pending_id.is_none(),
            "Previous near txs has not been signed"
        );
        require!(
            account.debt_info.is_none(),
            "The account has outstanding debt."
        );


        let config = self.internal_config();
        let near_gas_amount = if user_intention.intention.use_near_pay_gas {
            0
        } else {
            check_intention(&csna_id, config, &near_transactions_bytes)
        };
        require!(
            self.data().cur_available_near_gas >= near_gas_amount,
            "Insufficient cur_available_near_gas"
        );
        let protocol_fee_amount =
            config.near_tx_protocol_fee * user_intention.intention.near_transactions.len() as u128;
        let total_amount = near_gas_amount + protocol_fee_amount;
        if account.yoctonear_amount >= total_amount {
            self.internal_unwrap_mut_account(&csna_id).yoctonear_amount -= total_amount;
            self.data_mut().acc_distributed_near_gas += near_gas_amount;
            self.data_mut().cur_available_near_gas -= near_gas_amount;
            self.data_mut().acc_collected_protocol_fee += protocol_fee_amount;
            self.data_mut().cur_available_protocol_fee += protocol_fee_amount;
        } else {
            self.assert_first_tx_valid(
                &near_transactions_bytes[0],
                total_amount,
                &user_intention.intention,
            );
            self.internal_unwrap_mut_account(&csna_id).debt_info = Some(DebtInfo {
                near_gas_debt_amount: near_gas_amount,
                protocol_fee_debt_amount: protocol_fee_amount,
                relayer_debt_info: None,
            });
            self.data_mut().acc_distributed_near_gas += near_gas_amount;
            self.data_mut().cur_available_near_gas -= near_gas_amount;
            self.data_mut().cur_near_gas_debt += near_gas_amount;
            self.data_mut().cur_protocol_fee_debt += protocol_fee_amount;
        }
        self.internal_unwrap_mut_account(&csna_id).near_pending_id = Some(near_pending_id.clone());
        self.internal_unwrap_mut_account(&csna_id).nonce += 1;


        let need_signature_num = user_intention.intention.near_transactions.len();
        let near_pending_info = NEARPendingInfo {
            csna_id: csna_id.clone(),
            csna_path,
            near_pending_id: near_pending_id.clone(),
            near_transactions: near_transactions_bytes.clone(),
            signed_transactions: vec![None; need_signature_num],
            create_time_sec: nano_to_sec(env::block_timestamp()),
            last_sign_time_sec: 0,
        };
        self.internal_set_near_pending_info(&near_pending_id, near_pending_info);


        Event::GenerateNearPendingInfo {
            csna_id: &csna_id,
            near_pending_id: &near_pending_id,
        }
        .emit();


        if near_gas_amount > 0 {
            Promise::new(csna_id.clone()).transfer(NearToken::from_yoctonear(near_gas_amount));
        }
    }
```

Listing 2.5: user_intention.rs

```
    pub fn assert_first_tx_valid(
        &self,
        first_transaction_bytes: &[u8],
        total_amount: u128,
        intention: &Intention,
    ) {
        // It has been validated in the check_intention function, so it's safe to directly unwrap.
        let first_transaction =
            borsh::from_slice::<NearTransaction>(first_transaction_bytes).unwrap();
        require!(
            first_transaction.receiver_id() == &intention.gas_token,
            "first tx receiver_id does not match the intention.gas_token"
        );
        let gas_token_details =
            self.internal_unwrap_gas_token_details(first_transaction.receiver_id());
        let need_gas_token_amount = gas_token_details.to_gas_token_amount(total_amount);
        require!(
```

```rust
        need_gas_token_amount <= intention.gas_limit,
        "need_gas_token_amount > intention.gas_limit"
    );
    match &first_transaction.actions()[0] {
        Action::FunctionCall(call) => {
            require!(
                call.method_name == "ft_transfer_call",
                "first tx must be ft_transfer_call"
            );
            require!(call.deposit == 1, "first tx need yocto");
            require!(
                call.gas >= Gas::from_tgas(50).as_gas(),
                "first tx must be greater than 50T"
            );
            let ft_transfer_call_args = serde_json::from_slice::<CheckFirstTxArgs>(&call.args)
                .expect("first tx args invalid");
            require!(
                ft_transfer_call_args.receiver_id == env::current_account_id(),
                "first tx args.receiver_id invalid"
            );
            require!(
                ft_transfer_call_args.amount.0 >= need_gas_token_amount,
                format!(
                    "first tx args.amount must be greater than or equal to {}",
                    need_gas_token_amount
                )
            );
            let ft_transfer_call_msg =
                serde_json::from_str::<TokenReceiverMessage>(&ft_transfer_call_args.msg)
                    .expect("invalid msg");
            require!(
                matches!(ft_transfer_call_msg, TokenReceiverMessage::Deposit),
                "first tx args.msg invalid"
            );
        }
        _ => env::panic_str("first tx invalid"),
    }
}
```

Listing 2.6: user_intention.rs

Impact The contract may fail to collect protocol fees and incur gas fee losses due to bypassed debt settlement.

Suggestion Revise the logic to ensure users will actually pay the corresponding fees.

Note The updated implementation reduces the likelihood of this situation by checking whether the user's account has sufficient gas tokens before allowing the transaction to be signed on-chain. However, under certain edge cases, the issue described may still occur. The team is aware of this possibility and has accepted its occurrence.

2.1.4 Lack of updating global variables in function `csna_verify_deposit()`

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description The protocol allows users to deposit supported gas tokens as gas, which are then recorded in the user's internal account under `yoctonear_amount` based on the oracle's price. The deposited gas tokens are periodically withdrawn by the protocol, exchanged for `NEAR`, and stored in the protocol. At this point, the variable `cur_available_near_gas`, which tracks the available `NEAR` gas for the protocol, is updated.

However, in the function `csna_verify_deposit()`, when a user pays the gas fee with their internal account, the contract does not check whether `cur_available_near_gas` is sufficient to cover the gas fee. Additionally, it does not deduct the paid gas fee from `cur_available_near_gas` while also accumulating `acc_distributed_near_gas`, which is incorrect.

```rust
pub fn csna_verify_deposit(
    &mut self,
    deposit_msg: DepositMsg,
    tx_bytes: Vec<u8>,
    vout: usize,
    tx_block_blockhash: String,
    tx_index: u64,
    merkle_proof: Vec<String>,
) -> Promise {
    self.assert_contract_running();
    require!(
        env::prepaid_gas() == Gas::from_tgas(300),
        "Insufficient gas"
    );


    let extra_msg_used_amount = if let Some(extra_msg_str) = deposit_msg.extra_msg.as_ref() {
        self.handle_extra_msg(&deposit_msg.recipient_id, extra_msg_str)
    } else {
        0
    };


    let relayer_deposit = env::attached_deposit().as_yoctonear();
    let account = self.internal_unwrap_mut_account(&deposit_msg.recipient_id);
    if extra_msg_used_amount > account.yoctonear_amount {
        require!(
            extra_msg_used_amount == relayer_deposit,
            format!("Need deposit {} yoctonear", extra_msg_used_amount)
        );
        account.debt_info = Some(DebtInfo {
            near_gas_debt_amount: 0,
            protocol_fee_debt_amount: 0,
            relayer_debt_info: Some(RelayerDebtInfo {
                relayer_id: env::predecessor_account_id(),
                total_debt_amount: relayer_deposit,
                repay_debt_amount: 0,
            }),
        });
    } else {
        account.yoctonear_amount -= extra_msg_used_amount;
        if relayer_deposit > 0 {
            Promise::new(env::predecessor_account_id())
                .transfer(NearToken::from_yoctonear(relayer_deposit));
        }
    }
    ext_bridge::ext(self.internal_config().bridge_account_id.clone())
        .with_static_gas(GAS_FOR_BRIDGE_VERIFY_DEPOSIT)
        .verify_deposit(
            deposit_msg,
            tx_bytes,
            vout,
            tx_block_blockhash,
            tx_index,
            merkle_proof,
        )
}
```

Listing 2.7: bridge.rs

Impact The contract may not properly track or manage available `NEAR` gas, leading to incorrect gas fee handling and potential discrepancies in the protocol's available gas balance.

Suggestion Revise the logic to ensure that `cur_available_near_gas` and `acc_distributed_near_gas` are updated correctly.

2.1.5 Overwritten debt_info leading to incorrect user debt

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `csna_verify_deposit()`, a `relayer` can pay on behalf of a user with insufficient `NEAR` balance in their internal account and generate the corresponding `debt_info`. However, the newly generated `debt_info` overwrites the user's existing `debt_info`, which is incorrect.

```
#[payable]
pub fn csna_verify_deposit(
    &mut self,
    deposit_msg: DepositMsg,
    tx_bytes: Vec<u8>,
    vout: usize,
    tx_block_blockhash: String,
    tx_index: u64,
    merkle_proof: Vec<String>,
) - > Promise {
    self.assert_contract_running();
    require!(
        env::prepaid_gas() == Gas::from_tgas(300),
        "Insufficient gas"
    );


    let extra_msg_used_amount = if let Some(extra_msg_str) = deposit_msg.extra_msg.as_ref() {
        self.handle_extra_msg(&deposit_msg.recipient_id, extra_msg_str)
    } else {
        0
    };


    let relayer_deposit = env::attached_deposit().as_yoctonear();
    let account = self.internal_unwrap_mut_account(&deposit_msg.recipient_id);
    if extra_msg_used_amount > account.yoctonear_amount {
        require!(
            extra_msg_used_amount == relayer_deposit,
            format!("Need deposit {} yoctonear", extra_msg_used_amount)
        );
        account.debt_info = Some(DebtInfo {
            near_gas_debt_amount: 0,
            protocol_fee_debt_amount: 0,
            relayer_debt_info: Some(RelayerDebtInfo {
                relayer_id: env::predecessor_account_id(),
                total_debt_amount: relayer_deposit,
                repay_debt_amount: 0,
            }),
        });
    } else {
        account.yoctonear_amount -= extra_msg_used_amount;
        if relayer_deposit > 0 {
            Promise::new(env::predecessor_account_id())
                .transfer(NearToken::from_yoctonear(relayer_deposit));
        }
    }
    ext_bridge::ext(self.internal_config().bridge_account_id.clone())
        .with_static_gas(GAS_FOR_BRIDGE_VERIFY_DEPOSIT)
        .verify_deposit(
            deposit_msg,
            tx_bytes,
            vout,
            tx_block_blockhash,
            tx_index,
            merkle_proof,
        )
}
```

Listing 2.8: bridge.rs

Impact The amounts the user needs to repay for `near_gas_debt_amount` and `protocol_fee_debt_amount` are reset to zero, and the `relayer` may not receive the `NEAR` they paid before.

Suggestion Revise the logic to ensure that `debt_info` will not be overwritten.

Note After the fix, the function `csna_verify_deposit()` only incurs a relayer fee, distinguishing it from the gas and protocol fees generated by the function `request_sign_near_txs()`, thereby resolving the issue of `debt_info` being overwritten. However, the new implementation, by design, allows users to initiate transaction requests via the `request_sign_near_txs()` function even while owing relayer fees.

2.1.6 Lack of callback function to handle the return value of function `create_account()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description When a user uses the protocol for the first time, the protocol creates a corresponding `NEAR` account for the user based on their `btc_public_key`. In the function `create_account()`, the protocol records the generated `csna_id` in the contract and then creates the `NEAR` account for the user. Since creating the `NEAR` account is a cross-contract call and there is no callback function to handle the result of the cross-contract call, if the cross-contract call fails, the `csna_id` is not actually created on the `NEAR_Network`, but the protocol has already recorded this `csna_id`. In this case, the user cannot use this `csna_id` to perform transactions on `NEAR`, and because the `csna_id` and `btc_public_key` are one-to-one mapped, the user cannot use the same `btc_public_key` to create a `NEAR` account again.

```rust
pub fn handle_extra_msg(&mut self, csna_id: &AccountId, extra_msg_str: &str) -> u128 {
    if let Ok(extra_msg) = serde_json::from_str::<ExtraMsg>(extra_msg_str) {
        let mut total_amount = 0;
        if let Some(StorageDepositMsg {
            contract_id,
            deposit,
            registration_only,
        }) = &extra_msg.storage_deposit_msg
        {
            total_amount = *deposit;
            ext_storage_management::ext(contract_id.clone())
                .with_static_gas(GAS_FOR_STORAGE_DEPOSIT)
                .with_attached_deposit(NearToken::from_yoctonear(*deposit))
                .storage_deposit(Some(csna_id.clone()), *registration_only);
        }
        if let Some(btc_public_key) = &extra_msg.btc_public_key {
            if !self.check_account_exists(csna_id) {
                let csna_path = get_csna_path(btc_public_key);
                require!(
                    csna_id == &self.get_csna_id(&csna_path),
                    "deposit_msg.extra_msg.btc_public_key does not match deposit_msg.
                        recipient_id"
                );
                let initial_csna_balance = self.internal_config().initial_csna_balance;
                total_amount += initial_csna_balance;
                self.create_account(csna_id, &csna_path, initial_csna_balance);
            }
        }
        total_amount
    } else {
        0
    }
}
```

Listing 2.9: deposit_msg.rs

```
pub fn create_account(
    &mut self,
    csna_id: &AccountId,
    csna_path: &str,
    initial_csna_balance: u128,
) {
    self.internal_set_account(csna_id, Account::new(csna_id));
    let full_access_key = self.generate_near_public_key(csna_path);
    Promise::new(csna_id.clone())
        .create_account()
        .add_full_access_key(full_access_key)
        .transfer(near_sdk::NearToken::from_yoctonear(initial_csna_balance));
}
```

Listing 2.10: deposit_msg.rs

Impact If the account fails to be created, the user cannot generate the `csna` using the same `btc_public_key` again.

Suggestion Revise the logic to ensure that after the function `create_account()` fails, the protocol still allows the user to create a `csna` using the same `btc_public_key`.

2.1.7 Lack of check on contract_id and deposited amount in function `handle_extra_msg()`

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The function `csna_verify_deposit()` may deposit storage fees into a specified contract according to the information in `deposit_msg.extra_msg`.

However, neither the target contract nor the deposited amount is properly validated. Malicious users can specify a target contract under their control and provide an excessively large deposit amount. In this case, if a `relayer` attaches the corresponding amount of `NEAR`, this `NEAR` will be directly deposited into the user-controlled contract, resulting in a loss of the protocol.

```
pub fn csna_verify_deposit(
    &mut self,
    deposit_msg: DepositMsg,
    tx_bytes: Vec<u8>,
    vout: usize,
    tx_block_blockhash: String,
    tx_index: u64,
    merkle_proof: Vec<String>,
) -> Promise {
    self.assert_contract_running();
    require!(
        env::prepaid_gas() == Gas::from_tgas(300),
        "Insufficient gas"
    );


    let extra_msg_used_amount = if let Some(extra_msg_str) = deposit_msg.extra_msg.as_ref() {
        self.handle_extra_msg(&deposit_msg.recipient_id, extra_msg_str)
    } else {
        0
    };


    let relayer_deposit = env::attached_deposit().as_yoctonear();
    let account = self.internal_unwrap_mut_account(&deposit_msg.recipient_id);
    if extra_msg_used_amount > account.yoctonear_amount {
        require!(
            extra_msg_used_amount == relayer_deposit,
            format!("Need deposit {} yoctonear", extra_msg_used_amount)
        );
        account.debt_info = Some(DebtInfo {
            near_gas_debt_amount: 0,
            protocol_fee_debt_amount: 0,
            relayer_debt_info: Some(RelayerDebtInfo {
                relayer_id: env::predecessor_account_id(),
                total_debt_amount: relayer_deposit,
                repay_debt_amount: 0,
            }),
        });
    } else {
        account.yoctonear_amount -= extra_msg_used_amount;
        if relayer_deposit > 0 {
            Promise::new(env::predecessor_account_id())
                .transfer(NearToken::from_yoctonear(relayer_deposit));
        }
    }
    ext_bridge::ext(self.internal_config().bridge_account_id.clone())
        .with_static_gas(GAS_FOR_BRIDGE_VERIFY_DEPOSIT)
        .verify_deposit(
            deposit_msg,
            tx_bytes,
            vout,
            tx_block_blockhash,
            tx_index,
            merkle_proof,
        )
}
```

Listing 2.11: bridge.rs

```rust
pub fn handle_extra_msg(&mut self, csna_id: &AccountId, extra_msg_str: &str) -> u128 {
    if let Ok(extra_msg) = serde_json::from_str::<ExtraMsg>(extra_msg_str) {
        let mut total_amount = 0;
        if let Some(StorageDepositMsg {
            contract_id,
            deposit,
            registration_only,
        }) = &extra_msg.storage_deposit_msg
        {
            total_amount = *deposit;
            ext_storage_management::ext(contract_id.clone())
                .with_static_gas(GAS_FOR_STORAGE_DEPOSIT)
                .with_attached_deposit(NearToken::from_yoctonear(*deposit))
                .storage_deposit(Some(csna_id.clone()), *registration_only);
        }
        if let Some(btc_public_key) = &extra_msg.btc_public_key {
            if !self.check_account_exists(csna_id) {
                let csna_path = get_csna_path(btc_public_key);
                require!(
                    csna_id == &self.get_csna_id(&csna_path),
                    "deposit_msg.extra_msg.btc_public_key does not match deposit_msg.
                        recipient_id"
                );
                let initial_csna_balance = self.internal_config().initial_csna_balance;
                total_amount += initial_csna_balance;
                self.create_account(csna_id, &csna_path, initial_csna_balance);
            }
        }
        total_amount
    } else {
        0
    }
}
```

Listing 2.12: deposit_msg.rs

Impact The relayer may suffer a loss.

Suggestion Add a check to ensure that the variable `contract_id` and `deposit` in `extra_msg` is legal and cannot be maliciously manipulated.

Feedback from the project `Relayers` should include sufficient off-chain checks to ensure that before prepaying the `NEAR` required for users to perform pre-actions, users will repay a sufficient amount of `nBTC` in the post-actions.

2.2 Note

2.2.1 Potential centralization risk

Introduced by `Version 1`

Description In the current implementation, several privileged roles are set to govern and regulate the system-wide operation (e.g., parameter setting, pause/unpause and grant roles). Additionally, the `owner` also has the ability to upgrade contracts. If the private keys of them are lost or maliciously exploited, it could potentially lead to losses for users.

2.2.2 Potential delayed price due to untimely price updated from oracles

Introduced by `Version 1`

Description When users without a `NEAR` account initiate a `NEAR` transaction through the contract, they can choose a specified gas token to pay the gas fee. The contract will calculate and deduct the corresponding value of the gas token to pay `NEAR` as gas for the user. However, the price of the gas token needs to be updated by invoking the `update_gas_token_price()` function to pull from the corresponding oracle. This may cause delays in the price when users are paying with the gas token. Thus, the team needs to ensure that `update_gas_token_price()` is invoked promptly to maintain price accuracy.

2.2.3 No guarantee of transaction success on NEAR and refund protocol_fee on failure

Introduced by `Version 1`

Description Users can use the function `request_sign_near_txs()` to construct a transaction on the `NEAR_Network`. Once the transaction is fully signed, the user's `NEAR` transaction will be executed on-chain. It is important to note that the protocol does not ensure the successful execution of the user's transaction on the `NEAR` Network. If the transaction fails, the protocol fee paid by the user is not refunded.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS