

# Security Audit

## Report for Satoshi - Bridge and nBTC

**Date:** December 12, 2024 **Version:** 1.0

**Contact:** [contact@blocksec.com](mailto:contact@blocksec.com)

# Contents

<b>Chapter 1 Introduction</b>	<b>1</b>
1.1 About Target Contracts . . . . .	1
1.2 Disclaimer . . . . .	1
1.3 Procedure of Auditing . . . . .	1
1.3.1 Software Security . . . . .	2
1.3.2 DeFi Security . . . . .	2
1.3.3 NFT Security . . . . .	2
1.3.4 Additional Recommendation . . . . .	2
1.4 Security Model . . . . .	3
<b>Chapter 2 Findings</b>	<b>4</b>
2.1 DeFi Security . . . . .	5
2.1.1 Incorrect check in function <code>mint_callback()</code> . . . . .	5
2.1.2 Incorrect fee update due to wrong key retrieval of <code>IterableMap</code> values . . .	6
2.1.3 Lack of check in function <code>withdraw_rbf()</code> . . . . .	9
2.1.4 Potential DoS due to improper check in function <code>sign_btc_transaction_callback()</code>	11
2.1.5 Improper time check for RBF cancellation . . . . .	16
2.2 Additional Recommendation . . . . .	18
2.2.1 Redundant code . . . . .	18
2.2.2 Function name typo correction . . . . .	21
2.3 Note . . . . .	21
2.3.1 Potential centralization risk . . . . .	21
2.3.2 Potential occupation of protocol-managed UTXOs by malicious users due to improper global parameters . . . . .	22

## Report Manifest

Item	Description
Client	Satoshi
Target	Satoshi - Bridge and nBTC

## Version History

Version	Date	Description
1.0	December 12, 2024	First release

## Signature

**About BlockSec** BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at [Email](#), [Twitter](#) and [Medium](#).

# 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-Bridge <sup>1</sup> and nBTC <sup>2</sup>.

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-Bridge and nBTC	<a href="#">Version 1</a>	<a href="#">dd660a9eb5e8982e8b163928e787b8fd9e9830ab</a>
	<a href="#">Version 2</a>	<a href="#">ade294b1d58eac0b291f5d6cd19292ec1c366f42</a>

## 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.

---

<sup>1</sup><https://github.com/Near-Bridge-Lab/btc-bridge-contract/tree/main/contracts/bridge>

<sup>2</sup><https://github.com/Near-Bridge-Lab/btc-bridge-contract/tree/main/contracts/nbtc>

- **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<sup>3</sup> and Common Weakness Enumeration<sup>4</sup>. 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

Impact	High	High	Medium
	Low	Medium	Low
		High	Low
		Likelihood	

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.

<sup>3</sup>[https://owasp.org/www-community/OWASP\\_Risk\\_Rating\\_Methodology](https://owasp.org/www-community/OWASP_Risk_Rating_Methodology)

<sup>4</sup><https://cwe.mitre.org/>

## Chapter 2 Findings

In total, we find **five** potential issues. Besides, we also have **two** recommendations and **two** notes as follows:

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

ID	Severity	Description	Category	Status
1	High	Incorrect check in function <code>mint_callback()</code>	DeFi Security	Fixed
2	High	Incorrect fee update due to wrong key retrieval of IterableMap values	DeFi Security	Fixed
3	Medium	Lack of check in function <code>withdraw_rbf()</code>	DeFi Security	Fixed
4	Medium	Potential DoS due to improper check in function <code>sign_btc_transaction_callback()</code>	DeFi Security	Fixed
5	Medium	Improper time check for RBF cancellation	DeFi Security	Confirmed
6	-	Redundant code	Recommendation	Fixed
7	-	Function name typo correction	Recommendation	Fixed
8	-	Potential centralization risk	Note	-
9	-	Potential occupation of protocol-managed UTXOs by malicious users due to improper global parameters	Note	-

The details are provided in the following sections.

## 2.1 DeFi Security

### 2.1.1 Incorrect check in function `mint_callback()`

**Severity** High

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** The bridge contract will mint the corresponding amount of `nBTC` to users once their `BTC` deposit transaction is successfully validated. If the minting is successful, the callback function `mint_callback()` will then register an internal account for the receiver if the receiver is not registered yet. However, the function's check for whether the user is already registered is incorrect. The current implementation only registers the user if they already have an account, which is the opposite of what should happen.

```
43  #[private]
44  pub fn mint_callback(
45      &mut self,
46      recipient_id: AccountId,
47      mint_amount: U128,
48      protocol_fee: U128,
49      relayer_fee: U128,
50      pending_utxo_info: PendingUTXOInfo,
51  ) -> bool {
52      let is_success = is_promise_success();
53      if !is_success {
54          self.data_mut()
55              .verified_deposit_utxo
56              .remove(&pending_utxo_info.utxo_storage_key);
57      } else {
58          if self.check_account_exists(&recipient_id) {
59              self.internal_set_account(&recipient_id, Account::new(&recipient_id));
60          }
61          if protocol_fee.0 > 0 {
62              self.data_mut().acc_collected_protocol_fee += protocol_fee.0;
63              self.data_mut().cur_available_protocol_fee += protocol_fee.0;
64          }
65          Event::UtxoAdded {
66              utxo_storage_keys: vec![pending_utxo_info.utxo_storage_key.clone()],
67          }
68              .emit();
69          self.internal_set_utxo(&pending_utxo_info.utxo_storage_key, pending_utxo_info.utxo);
70      }
71      Event::VerifyDepositDetails {
72          recipient_id: &recipient_id,
73          mint_amount,
74          protocol_fee,
75          relayer_account_id: env::signer_account_id(),
76          relayer_fee,
```



```
77         success: is_success,
78     }
79     .emit();
80     is_success
81 }
```

**Listing 2.1:** mint.rs

**Impact** The execution result of the function may not align with expectations, and the `account` structure recorded in the `bridge` contract may be overwritten.

**Suggestion** Revise the logic to register the corresponding `account` when the `account` is not registered.

### 2.1.2 Incorrect fee update due to wrong key retrieval of `IterableMap` values

**Severity** High

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** The bridge contract permits privileged accounts to use the `cancel_withdraw()` function, enabling them to initiate a new transaction with a higher gas fee using the RBF (Replace-By-Fee) feature to cancel a user's pending withdrawal if it is delayed. This new transaction, designed to replace the original one, increases the likelihood of being processed sooner. If the user's remaining `BTC`, after the withdrawal fee, is insufficient to cover the increased gas fee, the protocol absorbs the additional cost using its own fees.

These extra gas fees are recorded in the `reserved_protocol_fee` field within the `btc_pending_info` structure associated with the new RBF transaction and are also added to the global `cur_reserved_protocol_fee`. This setup ensures the correct rollback of the state if the new transaction fails to execute. Once the transaction is successfully mined, the contract should reduce the `cur_reserved_protocol_fee` by the amount listed as `reserved_protocol_fee` in the `verify_cancel_withdraw_burn_callback()` function. However, a problem arises because the callback function mistakenly uses the `reserved_protocol_fee` from the original transaction (which is null), rather than from the successfully mined new RBF transaction. Consequently, the `cur_reserved_protocol_fee` is not accurately updated, leading to discrepancies.

```
16 pub fn internal_cancel_withdraw(
17     &mut self,
18     original_btc_pending_verify_id: String,
19     output: Vec<TxOut>,
20 ) -> String {
21     let original_tx_pending_info =
22         self.internal_unwrap_btc_pending_verify_info(&original_btc_pending_verify_id);
23     require!(
24         nano_to_sec(env::block_timestamp()) - original_tx_pending_info.create_time_sec
25         > self.internal_config().max_btc_tx_pending_sec,
26         "Please wait user rbf"
27     );
28     require!(
29         original_tx_pending_info.original_tx_id.is_none()
```

```
30         && original_tx_pending_info.pending_info_type == PendingInfoType::Withdraw,
31         "Not valid original tx"
32     );
33     let cancel_withdraw_rbf_psbt =
34         self.generate_psbt_from_original_psbt_and_new_output(original_tx_pending_info, output);
35
36
37     let mut btc_pending_info = init_rbf_btc_pending_info(original_tx_pending_info);
38     let (actual_received_amount, gas_fee) = self.check_cancel_withdraw_rbf_psbt_valid(
39         original_tx_pending_info,
40         &cancel_withdraw_rbf_psbt,
41     );
42
43
44     btc_pending_info.gas_fee = gas_fee;
45     btc_pending_info.burn_amount = gas_fee;
46     btc_pending_info.actual_received_amount = actual_received_amount;
47     btc_pending_info.pending_info_type = PendingInfoType::CancelWithdrawRbf;
48     // Ensure that the RBF transaction pays more gas than the previous transaction.
49     let additional_gas_amount = btc_pending_info
50         .gas_fee
51         .saturating_sub(original_tx_pending_info.max_gas_fee.unwrap().0);
52     require!(additional_gas_amount > 0, "No gas increase.");
53     let excess_gas_fee = gas_fee
54         .saturating_sub(btc_pending_info.transfer_amount - btc_pending_info.withdraw_fee);
55     if excess_gas_fee > 0 {
56         require!(
57             self.internal_config().owner_id == env::predecessor_account_id(),
58             "gas fee exceeds the user's balance, only the owner is allowed to cancel"
59         );
60         require!(
61             self.data().cur_available_protocol_fee >= excess_gas_fee,
62             "Insufficient protocol fee"
63         );
64         self.data_mut().cur_available_protocol_fee -= excess_gas_fee;
65         self.data_mut().cur_reserved_protocol_fee += excess_gas_fee;
66         btc_pending_info.reserved_protocol_fee = Some(U128(excess_gas_fee));
67     }
68     self.set_rbf_pending_info(
69         &original_btc_pending_verify_id,
70         btc_pending_info,
71         cancel_withdraw_rbf_psbt,
72     )
73 }
```

Listing 2.2: cancel\_withdraw.rs

```
150     #[private]
151     pub fn verify_cancel_withdraw_burn_callback(
152         &mut self,
153         tx_id: String,
154         btc_pending_info: BTCPendingInfo,
155         protocol_fee: U128,
```

```
156     relayer_fee: U128,
157 ) -> bool {
158     let is_success = is_promise_success();
159     let config = self.internal_config();
160     let refund = btc_pending_info
161         .transfer_amount
162         .saturating_sub(btc_pending_info.withdraw_fee + btc_pending_info.burn_amount);
163     let burn_event = Event::CancelWithdrawDetails {
164         account_id: &btc_pending_info.account_id.clone(),
165         burn_amount: btc_pending_info.burn_amount.into(),
166         protocol_fee,
167         relayer_account_id: env::signer_account_id(),
168         relayer_fee,
169         refund: refund.into(),
170         success: is_success,
171     };
172     if is_success {
173         let tx_bytes = btc_pending_info.tx_bytes_with_sign.as_ref().unwrap();
174         let transaction = bytes_to_btc_transaction(tx_bytes);
175         let withdraw_change_address = config.get_change_address();
176         let withdraw_change_script_pubkey = withdraw_change_address.script_pubkey();
177         let mut utxo_storage_keys = vec![];
178         for (index, output) in transaction.output.into_iter().enumerate() {
179             if withdraw_change_script_pubkey == output.script_pubkey {
180                 let utxo = UTXO {
181                     path: env::current_account_id().to_string(),
182                     tx_bytes: tx_bytes.clone(),
183                     vout: index,
184                     balance: output.value.to_sat(),
185                 };
186                 let utxo_storage_key = generate_utxo_storage_key(tx_id.clone(), index as u32);
187                 self.internal_set_utxo(&utxo_storage_key, utxo);
188                 utxo_storage_keys.push(utxo_storage_key);
189             }
190         }
191         // Cancel Withdraw is RBF, so original_tx_id must be Some.
192         let original_tx_id = btc_pending_info.original_tx_id.as_ref().unwrap();
193         let original_btc_pending_info =
194             self.internal_remove_btc_pending_verify_info(original_tx_id);
195         self.data_mut().rbf_txs.remove(original_tx_id);
196         self.internal_unwrap_mut_account(&btc_pending_info.account_id)
197             .btc_pending_verify_list
198             .remove(original_tx_id);
199         if protocol_fee.0 > 0 {
200             self.data_mut().acc_collected_protocol_fee += protocol_fee.0;
201             self.data_mut().cur_available_protocol_fee += protocol_fee.0;
202         }
203         if let Some(U128(reserved_protocol_fee)) =
204             original_btc_pending_info.reserved_protocol_fee
205         {
206             self.data_mut().cur_reserved_protocol_fee -= reserved_protocol_fee;
207             self.data_mut().acc_protocol_fee_for_gas += reserved_protocol_fee;
208         }
209     }
210 }
```

```
209         if refund > 0 {
210             self.internal_transfer_nbtc(&btc_pending_info.account_id, refund);
211         }
212         Event::UtxoAdded { utxo_storage_keys }.emit();
213     } else {
214         self.internal_set_btc_pending_verify_info(&tx_id, btc_pending_info);
215     }
216     burn_event.emit();
217     is_success
218 }
```

**Listing 2.3:** burn.rs

**Impact** The global variable `cur_reserved_protocol_fee` is not correctly updated.

**Suggestion** Retrieve the correct `reserved_protocol_fee` from the `btc_pending_info` of the successfully mined RBF transaction.

### 2.1.3 Lack of check in function `withdraw_rbf()`

**Severity** Medium

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** Users can initiate an RBF to increase the gas fee and improve the chances of their withdrawal transaction being mined by calling the `withdraw_rbf()` function. Privileged accounts, on the other hand, can cancel a withdrawal operation for a transaction that has been stuck off-chain for too long using the `cancel_withdraw()` function. However, based on the current implementation, after the `cancel_withdraw()` function is executed, users are still allowed to call `withdraw_rbf()` to initiate another RBF in an attempt to complete the withdrawal. In this case, if the transaction is successfully confirmed, the increased gas fee is actually wasted, which does not align with the intended design flow.

```
119 pub fn withdraw_rbf(&mut self, original_btc_pending_verify_id: String, output: Vec<TxOut>) {
120     self.assert_contract_running();
121     let account_id = env::predecessor_account_id();
122     require!(
123         self.internal_unwrap_account(&account_id)
124             .btc_pending_sign_id
125             .is_none(),
126         "Previous btc tx has not been signed"
127     );
128     let btc_pending_sign_id =
129         self.internal_withdraw_rbf(&account_id, original_btc_pending_verify_id, output);
130     self.internal_unwrap_mut_account(&account_id)
131         .btc_pending_sign_id = Some(btc_pending_sign_id.clone());
132     Event::GenerateBtcPendingSignInfo {
133         account_id: &account_id,
134         btc_pending_sign_id: &btc_pending_sign_id,
135     }
136     .emit();
137 }
```

### Listing 2.4: bridge.rs

```
190 pub fn cancel_withdraw(&mut self, original_btc_pending_verify_id: String, output: Vec<TxOut>)
    {
191     self.assert_contract_running();
192     assert_one_yocto();
193     self.assert_owner_or_operators();
194     let user_account_id = self
195         .internal_unwrap_btc_pending_verify_info(&original_btc_pending_verify_id)
196         .account_id
197         .clone();
198     require!(
199         self.internal_unwrap_account(&user_account_id)
200             .btc_pending_sign_id
201             .is_none(),
202         "Assisted user previous btc tx has not been signed"
203     );
204     let btc_pending_sign_id =
205         self.internal_cancel_withdraw(original_btc_pending_verify_id, output);
206     self.internal_unwrap_mut_account(&user_account_id)
207         .btc_pending_sign_id = Some(btc_pending_sign_id.clone());
208     Event::GenerateBtcPendingSignInfo {
209         account_id: &user_account_id,
210         btc_pending_sign_id: &btc_pending_sign_id,
211     }
212     .emit();
213 }
```

### Listing 2.5: bridge.rs

```
16 pub fn internal_cancel_withdraw(
17     &mut self,
18     original_btc_pending_verify_id: String,
19     output: Vec<TxOut>,
20 ) -> String {
21     let original_tx_pending_info =
22         self.internal_unwrap_btc_pending_verify_info(&original_btc_pending_verify_id);
23     require!(
24         nano_to_sec(env::block_timestamp()) - original_tx_pending_info.create_time_sec
25         > self.internal_config().max_btc_tx_pending_sec,
26         "Please wait user rbf"
27     );
28     require!(
29         original_tx_pending_info.original_tx_id.is_none()
30         && original_tx_pending_info.pending_info_type == PendingInfoType::Withdraw,
31         "Not valid original tx"
32     );
33     let cancel_withdraw_rbf_psbt =
34         self.generate_psbt_from_original_psbt_and_new_output(original_tx_pending_info, output);
35
36
37     let mut btc_pending_info = init_rbf_btc_pending_info(original_tx_pending_info);
```

```
38     let (actual_received_amount, gas_fee) = self.check_cancel_withdraw_rbf_psbt_valid(
39         original_tx_pending_info,
40         &cancel_withdraw_rbf_psbt,
41     );
42
43
44     btc_pending_info.gas_fee = gas_fee;
45     btc_pending_info.burn_amount = gas_fee;
46     btc_pending_info.actual_received_amount = actual_received_amount;
47     btc_pending_info.pending_info_type = PendingInfoType::CancelWithdrawRbf;
48     // Ensure that the RBF transaction pays more gas than the previous transaction.
49     let additional_gas_amount = btc_pending_info
50         .gas_fee
51         .saturating_sub(original_tx_pending_info.max_gas_fee.unwrap().0);
52     require!(additional_gas_amount > 0, "No gas increase.");
53     let excess_gas_fee = gas_fee
54         .saturating_sub(btc_pending_info.transfer_amount - btc_pending_info.withdraw_fee);
55     if excess_gas_fee > 0 {
56         require!(
57             self.internal_config().owner_id == env::predecessor_account_id(),
58             "gas fee exceeds the user's balance, only the owner is allowed to cancel"
59         );
60         require!(
61             self.data().cur_available_protocol_fee >= excess_gas_fee,
62             "Insufficient protocol fee"
63         );
64         self.data_mut().cur_available_protocol_fee -= excess_gas_fee;
65         self.data_mut().cur_reserved_protocol_fee += excess_gas_fee;
66         btc_pending_info.reserved_protocol_fee = Some(U128(excess_gas_fee));
67     }
68     self.set_rbf_pending_info(
69         &original_btc_pending_verify_id,
70         btc_pending_info,
71         cancel_withdraw_rbf_psbt,
72     )
73 }
```

**Listing 2.6:** cancel\_withdraw.rs

**Impact** The increased gas fee caused by the operation of cancel withdrawal may be wasted.

**Suggestion** Add a check to ensure that after invoking `cancel_withdraw()`, the user cannot invoke `withdraw_rbf()` to accelerate the original transaction.

#### 2.1.4 Potential DoS due to improper check in function

`sign_btc_transaction_callback()`

**Severity** Medium

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** The function `withdraw_rbf()` allows users to employ the RBF (Replace-by-Fee)

feature to increase the gas fee and resend a transaction if the original withdrawal transaction becomes delayed. However, an issue arises during the chain signing process for the RBF transaction if the original transaction has already been mined and confirmed. In such cases, the callback function `sign_btc_transaction_callback()` encounters an error after processing the successful chain sign result.

Specifically, once the withdrawal transaction is successfully verified, the contract removes the corresponding `original_tx_id` from `btc_pending_sign_txs`. The `sign_btc_transaction_callback()` function then checks for the existence of the `original_tx_id` associated with the RBF transaction in `btc_pending_sign_txs`. If it no longer exists, the function will revert the transaction. This reversion prevents the removal of the corresponding `btc_pending_sign_id` from the user's account information, consequently blocking further actions by the user, such as additional withdrawals. This can inadvertently lead to a denial of service (DoS) for the affected user.

```
15 fn ft_on_transfer(
16     &mut self,
17     sender_id: AccountId,
18     amount: U128,
19     msg: String,
20 ) -> PromiseOrValue<U128> {
21     self.assert_contract_running();
22     let amount = amount.into();
23     require!(
24         amount >= self.internal_config().min_withdraw_amount,
25         "Invalid amount"
26     );
27     let message = serde_json::from_str::<TokenReceiverMessage>(&msg).expect("INVALID MSG");
28     let token_id = env::predecessor_account_id();
29     match message {
30         TokenReceiverMessage::Withdraw {
31             target_btc_address,
32             input,
33             output,
34         } => {
35             require!(
36                 token_id == self.internal_config().nbtch_account_id,
37                 "Not Allow"
38             );
39             let (psbt, utxo_storage_keys, vutxos) =
40                 self.generate_psbt_and_vutxos(input, output);
41             require!(
42                 self.internal_unwrap_or_create_mut_account(&sender_id)
43                     .btc_pending_sign_id
44                     .is_none(),
45                 "Previous btc tx has not been signed"
46             );
47             let target_address_script_pubkey =
48                 string_to_btc_address(&target_btc_address).script_pubkey();
49
50
51             let withdraw_change_address_script_pubkey =
```

```
52         self.internal_config().get_change_address().script_pubkey();
53     let withdraw_fee = self.internal_config().withdraw_bridge_fee.get_fee(amount);
54     let (actual_received_amount, gas_fee) = self.check_withdraw_psbtx_valid(
55         &target_address_script_pubkey,
56         &withdraw_change_address_script_pubkey,
57         &psbt,
58         &vutxos,
59         amount,
60         withdraw_fee,
61     );
62
63
64     let need_signature_num = psbt.unsigned_tx.input.len();
65     let psbt_hex = psbt.serialize_hex();
66     let btc_pending_sign_id = psbt.extract_tx().unwrap().compute_txid().to_string();
67     let btc_pending_info = BTCPendingInfo {
68         account_id: sender_id.clone(),
69         btc_pending_sign_id: btc_pending_sign_id.clone(),
70         transfer_amount: amount,
71         actual_received_amount,
72         withdraw_fee,
73         gas_fee,
74         burn_amount: amount - withdraw_fee,
75         psbt_hex,
76         vutxos,
77         signatures: vec![None; need_signature_num],
78         tx_bytes_with_sign: None,
79         create_time_sec: nano_to_sec(env::block_timestamp()),
80         last_sign_time_sec: 0,
81         original_tx_id: None,
82         max_gas_fee: Some(U128(gas_fee)),
83         pending_info_type: PendingInfoType::Withdraw,
84         reserved_protocol_fee: None,
85     };
86     require!(
87         self.data_mut()
88             .btc_pending_sign_txs
89             .insert(btc_pending_sign_id.clone(), btc_pending_info.into())
90             .is_none(),
91         "Already in pending sign"
92     );
93     self.internal_unwrap_mut_account(&sender_id)
94         .btc_pending_sign_id = Some(btc_pending_sign_id.clone());
95     Event::UtxoRemoved { utxo_storage_keys }.emit();
96     Event::GenerateBtcPendingSignInfo {
97         account_id: &sender_id,
98         btc_pending_sign_id: &btc_pending_sign_id,
99     }
100     .emit();
101     PromiseOrValue::Value(U128(0))
102 }
103 }
104 }
```



Listing 2.7: token\_receiver.rs

```
16 pub fn sign_btc_transaction(  
17     &mut self,  
18     btc_pending_sign_id: String,  
19     sign_index: usize,  
20     key_version: u32,  
21 ) -> Promise {  
22     self.assert_contract_running();  
23     let btc_pending_info = self.internal_unwrap_btc_pending_sign_info(&btc_pending_sign_id);  
24     require!(  
25         btc_pending_info.signatures[sign_index].is_none(),  
26         "Already signed"  
27     );  
28     let payload = get_hash_to_sign(&btc_pending_info.get_psbt(), sign_index);  
29     let path = btc_pending_info.vutxos[sign_index].get_path();  
30     self.internal_sign_btc_transaction(  
31         btc_pending_info.account_id.clone(),  
32         btc_pending_sign_id,  
33         sign_index,  
34         payload,  
35         path,  
36         key_version,  
37     )  
38 }
```

Listing 2.8: chain\_signatures.rs

```
72 pub fn internal_sign_btc_transaction(  
73     &mut self,  
74     account_id: AccountId,  
75     btc_pending_sign_id: String,  
76     sign_index: usize,  
77     payload: [u8; 32],  
78     path: String,  
79     key_version: u32,  
80 ) -> Promise {  
81     self.sign_promise(SignRequest {  
82         payload,  
83         path,  
84         key_version,  
85     })  
86     .then(  
87         Self::ext(env::current_account_id())  
88         .with_static_gas(GAS_FOR_SIGN_BTC_TRANSACTION_CALL_BACK)  
89         .sign_btc_transaction_callback(account_id, btc_pending_sign_id, sign_index),  
90     )  
91 }
```

Listing 2.9: chain\_signatures.rs

```
113
114 pub fn sign_btc_transaction_callback(
115     &mut self,
116     account_id: AccountId,
117     btc_pending_sign_id: String,
118     sign_index: usize,
119 ) -> bool {
120     if let Some(result_bytes) = promise_result_as_success() {
121         let signature = serde_json::from_slice::<SignatureResponse>(&result_bytes)
122             .expect("Invalid signature");
123         let mut btc_pending_info =
124             self.internal_remove_btc_pending_sign_info(&btc_pending_sign_id);
125         if let Some(original_tx_id) = btc_pending_info.original_tx_id.as_ref() {
126             require!(
127                 self.check_btc_pending_verify_info_exists(original_tx_id),
128                 "The tx has been verified"
129             );
130         }
131         require!(
132             btc_pending_info.signatures[sign_index].is_none(),
133             "Already signed"
134         );
135         btc_pending_info.signatures[sign_index] = Some(signature.clone());
136         btc_pending_info.last_sign_time_sec = nano_to_sec(env::block_timestamp());
137         Event::BtcInputSignature {
138             account_id: &account_id,
139             btc_pending_sign_id: &btc_pending_sign_id,
140             sign_index,
141             signature: &signature,
142         }
143         .emit();
144         let mut psbt = btc_pending_info.get_psbt();
145         psbt.inputs[sign_index].final_script_witness = Some(Witness::p2wpkh(
146             &signature.to_btc_signature(),
147             &self
148                 .generate_btc_public_key(&btc_pending_info.vutxos[sign_index].get_path())
149                 .inner,
150         ));
151         btc_pending_info.psbt_hex = psbt.serialize_hex();
152         if btc_pending_info.is_all_signed() {
153             let transaction = psbt.extract_tx().expect("extract_tx failed");
154             let tx_bytes_with_sign = serialize(&transaction);
155             let tx_id = transaction.compute_txid().to_string();
156             Event::SignedBtcTransaction {
157                 account_id: &account_id,
158                 tx_id: tx_id.clone(),
159                 tx_bytes: &tx_bytes_with_sign,
160             }
161             .emit();
162             btc_pending_info.tx_bytes_with_sign = Some(tx_bytes_with_sign);
163             let account = self.internal_unwrap_mut_account(&account_id);
164             let clear_account_btc_pending_sign_id =
```

```
165         account.btc_pending_sign_id.take() == Some(btc_pending_sign_id);
166         require!(clear_account_btc_pending_sign_id, "Internal error");
167         if btc_pending_info.original_tx_id.is_none() {
168             account.btc_pending_verify_list.insert(tx_id.clone());
169         }
170         self.internal_set_btc_pending_verify_info(&tx_id, btc_pending_info);
171     } else {
172         self.internal_set_btc_pending_sign_info(&btc_pending_sign_id, btc_pending_info);
173     }
174     true
175 } else {
176     false
177 }
178 }
```

**Listing 2.10:** chain\_signatures.rs

**Impact** The user may be unable to transfer [BTC](#) back from the [NEAR\\_Network](#) to the [BTC\\_Network](#).

**Suggestion** Add a method that, once the original transaction has been confirmed on-chain, allows for the deletion of the pending sign of the [RBF](#) transaction corresponding to the original transaction.

### 2.1.5 Improper time check for RBF cancellation

**Severity** Medium

**Status** Confirmed

**Introduced by** [Version 1](#)

**Description** If a user's withdrawal transaction has been stuck for a period of time, privileged accounts are allowed to utilize the [RBF](#) feature to cancel the user's withdrawal transaction via the function `cancel_withdraw()`. However, the time check is not correct.

Specifically, it only checks whether the creation time of the user's original withdrawal transaction has exceeded the limit, without considering that the user may have re-submitted a transaction with increased gas fee in the meantime. Given that the gas fee increases with each new transaction submission, if the user has just submitted an [RBF](#) transaction and it gets canceled immediately, the gas fee will be wasted.

```
190 pub fn cancel_withdraw(&mut self, original_btc_pending_verify_id: String, output: Vec<TxOut>)
191 {
192     self.assert_contract_running();
193     assert_one_yocto();
194     self.assert_owner_or_operators();
195     let user_account_id = self
196         .internal_unwrap_btc_pending_verify_info(&original_btc_pending_verify_id)
197         .account_id
198         .clone();
199     require!(
200         self.internal_unwrap_account(&user_account_id)
201             .btc_pending_sign_id
202             .is_none(),
```

```
202         "Assisted user previous btc tx has not been signed"
203     );
204     let btc_pending_sign_id =
205         self.internal_cancel_withdraw(original_btc_pending_verify_id, output);
206     self.internal_unwrap_mut_account(&user_account_id)
207         .btc_pending_sign_id = Some(btc_pending_sign_id.clone());
208     Event::GenerateBtcPendingSignInfo {
209         account_id: &user_account_id,
210         btc_pending_sign_id: &btc_pending_sign_id,
211     }
212     .emit();
213 }
```

Listing 2.11: bridge.rs

```
16 pub fn internal_cancel_withdraw(
17     &mut self,
18     original_btc_pending_verify_id: String,
19     output: Vec<TxOut>,
20 ) -> String {
21     let original_tx_pending_info =
22         self.internal_unwrap_btc_pending_verify_info(&original_btc_pending_verify_id);
23     require!(
24         nano_to_sec(env::block_timestamp()) - original_tx_pending_info.create_time_sec
25         > self.internal_config().max_btc_tx_pending_sec,
26         "Please wait user rbf"
27     );
28     require!(
29         original_tx_pending_info.original_tx_id.is_none()
30         && original_tx_pending_info.pending_info_type == PendingInfoType::Withdraw,
31         "Not valid original tx"
32     );
33     let cancel_withdraw_rbf_psbt =
34         self.generate_psbt_from_original_psbt_and_new_output(original_tx_pending_info, output);
35
36
37     let mut btc_pending_info = init_rbf_btc_pending_info(original_tx_pending_info);
38     let (actual_received_amount, gas_fee) = self.check_cancel_withdraw_rbf_psbt_valid(
39         original_tx_pending_info,
40         &cancel_withdraw_rbf_psbt,
41     );
42
43
44     btc_pending_info.gas_fee = gas_fee;
45     btc_pending_info.burn_amount = gas_fee;
46     btc_pending_info.actual_received_amount = actual_received_amount;
47     btc_pending_info.pending_info_type = PendingInfoType::CancelWithdrawRbf;
48     // Ensure that the RBF transaction pays more gas than the previous transaction.
49     let additional_gas_amount = btc_pending_info
50         .gas_fee
51         .saturating_sub(original_tx_pending_info.max_gas_fee.unwrap().0);
52     require!(additional_gas_amount > 0, "No gas increase.");
53     let excess_gas_fee = gas_fee
```

```
54     .saturating_sub(btc_pending_info.transfer_amount - btc_pending_info.withdraw_fee);
55     if excess_gas_fee > 0 {
56         require!(
57             self.internal_config().owner_id == env::predecessor_account_id(),
58             "gas fee exceeds the user's balance, only the owner is allowed to cancel"
59         );
60         require!(
61             self.data().cur_available_protocol_fee >= excess_gas_fee,
62             "Insufficient protocol fee"
63         );
64         self.data_mut().cur_available_protocol_fee -= excess_gas_fee;
65         self.data_mut().cur_reserved_protocol_fee += excess_gas_fee;
66         btc_pending_info.reserved_protocol_fee = Some(U128(excess_gas_fee));
67     }
68     self.set_rbf_pending_info(
69         &original_btc_pending_verify_id,
70         btc_pending_info,
71         cancel_withdraw_rbf_psbt,
72     )
73 }
```

**Listing 2.12:** cancel\_withdraw

**Impact** This issue may lead to unnecessary gas fee waste if a user's recently submitted [RBF](#) is canceled prematurely.

**Suggestion** The timing check should be updated to account for any new [RBF](#) submissions by the user, ensuring that the cancellation only occurs after a reasonable period following the most recent [RBF](#).

**Feedback from the project** The creation time of the withdrawal is chosen because the contract cannot verify whether the [RBF](#) is valid. If the user increases the gas fee by only 1 [Satoshi](#) each time, they can occupy the [UTXO](#) for an extended period until the network fee rate drops.

## 2.2 Additional Recommendation

### 2.2.1 Redundant code

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** In the function `internal_verify_deposit()`, the second parameter of `check_deposit_msg()` can directly use `mint_amount` instead of recalculating it. Similarly, in the function `sign_btc_transaction_callback()`, the computation of `tx_id` does not include `final_script_witness`, so it doesn't need to be calculated again. The parameter `btc_pending_sign_id` can be used directly.

```
7     pub fn internal_verify_deposit(
8         &mut self,
9         deposit_amount: u128,
10        tx_block_blockhash: String,
11        tx_index: u64,
```

```
12     merkle_proof: Vec<String>,
13     pending_utxo_info: PendingUTXOInfo,
14     deposit_msg: DepositMsg,
15 ) -> Promise {
16     let config = self.internal_config();
17     let recipient_id = deposit_msg.recipient_id.clone();
18     let confirmations = self.get_confirmations(config, deposit_amount);
19     let promise = self.verify_transaction_inclusion_promise(
20         config.btc_light_client_account_id.clone(),
21         pending_utxo_info.tx_id.clone(),
22         tx_block_blockhash,
23         tx_index,
24         merkle_proof,
25         confirmations,
26     );
27
28
29     if deposit_amount < config.min_deposit_amount {
30         promise.then(
31             Self::ext(env::current_account_id())
32                 .with_static_gas(GAS_FOR_UNAVAILABLE_UTXO_CALL_BACK)
33                 .unavailable_utxo_callback(recipient_id, pending_utxo_info),
34         )
35     } else {
36         let deposit_fee = config.deposit_bridge_fee.get_fee(deposit_amount);
37         let mint_amount = deposit_amount - deposit_fee;
38         let (protocol_fee, relayer_fee) = config
39             .deposit_bridge_fee
40             .get_protocol_and_relayer_fee(deposit_fee);
41
42
43         let post_actions = self.check_deposit_msg(deposit_msg, deposit_amount - deposit_fee);
44         promise.then(
45             Self::ext(env::current_account_id())
46                 .with_static_gas(GAS_FOR_VERIFY_DEPOSIT_CALL_BACK)
47                 .verify_deposit_callback(
48                     recipient_id,
49                     mint_amount.into(),
50                     protocol_fee.into(),
51                     relayer_fee.into(),
52                     pending_utxo_info,
53                     post_actions,
54                 ),
55         )
56     }
57 }
```

**Listing 2.13:** deposit.rs

```
113 pub fn sign_btc_transaction_callback(
114     &mut self,
115     account_id: AccountId,
116     btc_pending_sign_id: String,
```

```
117     sign_index: usize,
118 ) -> bool {
119     if let Some(result_bytes) = promise_result_as_success() {
120         let signature = serde_json::from_slice::<SignatureResponse>(&result_bytes)
121             .expect("Invalid signature");
122         let mut btc_pending_info =
123             self.internal_remove_btc_pending_sign_info(&btc_pending_sign_id);
124         if let Some(original_tx_id) = btc_pending_info.original_tx_id.as_ref() {
125             require!(
126                 self.check_btc_pending_verify_info_exists(original_tx_id),
127                 "The tx has been verified"
128             );
129         }
130         require!(
131             btc_pending_info.signatures[sign_index].is_none(),
132             "Already signed"
133         );
134         btc_pending_info.signatures[sign_index] = Some(signature.clone());
135         btc_pending_info.last_sign_time_sec = nano_to_sec(env::block_timestamp());
136         Event::BtcInputSignature {
137             account_id: &account_id,
138             btc_pending_sign_id: &btc_pending_sign_id,
139             sign_index,
140             signature: &signature,
141         }
142         .emit();
143         let mut psbt = btc_pending_info.get_psbt();
144         psbt.inputs[sign_index].final_script_witness = Some(Witness::p2wpkh(
145             &signature.to_btc_signature(),
146             &self
147                 .generate_btc_public_key(&btc_pending_info.vutxos[sign_index].get_path())
148                 .inner,
149         ));
150         btc_pending_info.psbt_hex = psbt.serialize_hex();
151         if btc_pending_info.is_all_signed() {
152             let transaction = psbt.extract_tx().expect("extract_tx failed");
153             let tx_bytes_with_sign = serialize(&transaction);
154             let tx_id = transaction.compute_txid().to_string();
155             Event::SignedBtcTransaction {
156                 account_id: &account_id,
157                 tx_id: tx_id.clone(),
158                 tx_bytes: &tx_bytes_with_sign,
159             }
160             .emit();
161             btc_pending_info.tx_bytes_with_sign = Some(tx_bytes_with_sign);
162             let account = self.internal_unwrap_mut_account(&account_id);
163             let clear_account_btc_pending_sign_id =
164                 account.btc_pending_sign_id.take() == Some(btc_pending_sign_id);
165             require!(clear_account_btc_pending_sign_id, "Internal error");
166             if btc_pending_info.original_tx_id.is_none() {
167                 account.btc_pending_verify_list.insert(tx_id.clone());
168             }
169             self.internal_set_btc_pending_verify_info(&tx_id, btc_pending_info);
```

```
170         } else {
171             self.internal_set_btc_pending_sign_info(&btc_pending_sign_id, btc_pending_info);
172         }
173         true
174     } else {
175         false
176     }
177 }
```

Listing 2.14: chain\_signature.rs

**Suggestion** Remove redundant code.

## 2.2.2 Function name typo correction

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** The function `remote_vutxo_by_psbt()` is responsible for processing a partially signed Bitcoin transaction (PSBT). It iterates through the inputs of the PSBT, retrieves the corresponding UTXOs from storage, and returns a tuple containing the keys of the UTXOs removed and the UTXOs themselves. However, the function name contains a typo: "remote" should likely be "remove" to better reflect its purpose of removing UTXOs from storage.

```
58 pub fn remote_vutxo_by_psbt(&mut self, psbt: &Psbt) -> (Vec<String>, Vec<VUTXO>) {
59     let mut utxo_storage_keys = vec![];
60     let vutxos = psbt
61         .unsigned_tx
62         .input
63         .clone()
64         .into_iter()
65         .map(|input| {
66             let utxo_storage_key = out_point_to_utxo_storage_key(&input.previous_output);
67             utxo_storage_keys.push(utxo_storage_key.clone());
68             self.data_mut()
69                 .utxos
70                 .remove(&utxo_storage_key)
71                 .unwrap_or_else(|| panic!("UTXO {} not exist", utxo_storage_key))
72         })
73         .collect::<Vec<_>>();
74     (utxo_storage_keys, vutxos)
75 }
```

Listing 2.15: utxo.rs

**Suggestion** Use the function name `remove_vutxo_by_psbt()` instead of `remote_vutxo_by_psbt()`.

## 2.3 Note

### 2.3.1 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.

### 2.3.2 Potential occupation of protocol-managed UTXOs by malicious users due to improper global parameters

**Introduced by** `Version 1`

**Description** The function `check_withdraw_psbtc_valid()` is used to verify the validity of a user-submitted `PSBT`. Within the function, the number of `UTXOs` managed by the protocol is retrieved and checked. If this number exceeds a certain threshold, it indicates that the protocol's `UTXOs` need to be consolidated or split, and further validation of the submitted `PSBT` is required.

It is important to note that the protocol sets the `max_change_number` and `max_withdrawal_input_number` parameters to limit the `PSBT` submitted by users. Please ensure that these parameters are assigned reasonable values to reduce the risk of malicious users occupying the `UTXOs` managed by the protocol.

