Security Audit Report for XP-Near-Integration

Date: Jan 12, 2023

Version: 2.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.01.13 14:32:05 +08'00'

Contents

1 Introduction **1**

1.1 About Target Contracts . . . 1

1.2 Disclaimer . . . 1

1.3 Procedure of Auditing . . . 2

1.3.1 Software Security . . . 2

1.3.2 DeFi Security . . . 2

1.3.3 NFT Security . . . 3

1.3.4 Additional Recommendation . . . 3

1.4 Security Model . . . 3

2 Findings **4**

2.1 Software Security . . . 4

2.1.1 Lack of Failure Handle in Callback Functions . . . 4

2.2 DeFi Security . . . 5

2.2.1 Lack of Sanity Check in withdraw_nft() . . . 5

2.2.2 Lack of Specified Gas Distribution in Cross Contract Invocation . . . 8

2.2.3 Improper Calculation for Storage Balance . . . 11

2.2.4 Lack of Whitelist Check in freeze_nft() . . . 12

2.2.5 Incorrect tx_fee Receiver . . . 12

2.3 Additional Recommendation . . . 14

2.3.1 Redundant Usage of Macro payable . . . 14

2.3.2 Improper Usage of the Data Structures . . . 18

2.3.3 Inconsistent Type of Input in validate_whitelist() . . . 19

2.3.4 Improper Use of Private Function . . . 19

2.3.5 Potential Centralization Problem . . . 20

2.3.6 Lack of Check of Updated group_key . . . 20

2.4 Notes . . . 21

2.4.1 Secure Implementation of the Offchain Mechanism . . . 21

2.4.2 Fee Amount Calculation . . . 21

Report Manifest

Item	Description
Client	XP NETWORK
Target	XP-Near-Integration

Version History

Version	Date	Description
1.0	December 20, 2022	First Version
2.0	January 12, 2023	Second Version

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 xp-near-integration [1].

The auditing process is iterative. Specifically, we will 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. Our audit report is responsible for the only initial version (i.e., `Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project		Commit SHA
XP-Near-Integration	`Version 1`	`f26d3696e33eec4ed8aca3604a6848274b25a347`
	`Version 2`	`436edf9a6ea6fbf62bb21cfc4b258fd6d10dda7c`
	`Version 3`	`b7b4cc3a1962093e5b621edb5e6feb11f9d36b5f`

Note that, we did **NOT** audit all the modules in the repository. The modules covered by this audit report include **xp-near-integration/contract/xpbridge/src** folder contract only. Specifically, the files covered in this audit include:

- event.rs
- external.rs
- lib.rs

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.

[1] https://github.com/XP-NETWORK/xp-near-integration

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.

- **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
∗ Access control
∗ 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 **six** potential issues. We also have **six** recommendations and **two** notes as follows:

- High Risk: 5
- Medium Risk: 0
- Low Risk: 1
- Recommendations: 6
- Notes: 2

ID	Severity	Description	Category	Status
1	High	Lack of Failure Handle in Callback Functions	Software Security	Fixed
2	High	Lack of Sanity Check in withdraw_nft()	DeFi Security	Fixed
3	High	Lack of Specified Gas Distribution in Cross Contract Invocation	DeFi Security	Fixed
4	High	Improper Calculation for Storage Balance	DeFi Security	Fixed
5	Low	Lack of Whitelist Check in freeze_nft()	DeFi Security	Fixed
6	High	Incorrect tx_fee Receiver	DeFi Security	Fixed
7	-	Redundant Usage of Macro payable	Recommendation	Fixed
8	-	Improper Usage of the Data Structures	Recommendation	Fixed
9	-	Inconsistent Type of Input in validate_whitelist()	Recommendation	Fixed
10	-	Improper Use of Private Function	Recommendation	Fixed
11	-	Potential Centralization Problem	Recommendation	Confirmed
12	-	Lack of Check of Updated group_key	Recommendation	Fixed
13	-	Secure Implementation of the Offchain Mechanism	Note	Confirmed
14	-	Fee Amount Calculation	Note	Confirmed

The details are provided in the following sections.

2.1 Software Security

2.1.1 Lack of Failure Handle in Callback Functions

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Potential failure of the cross-contract invocations can not be handled in the current implementation.

Take the function `validate_unfreeze_nft()` as an example, if the transfer of the unfreezed `NFT` is failed, there is no logic to make sure the user would receive the `NFT`.

```
    #[payable]
    pub fn validate_unfreeze_nft(
        &mut self,
        data: UnfreezeNftData,
        sig_data: Vec<u8>,
) -> Promise {
    require!(!self.paused, "paused");

    self.require_sig(
        data.action_id.into(),
        data.try_to_vec().unwrap(),
        sig_data,
        b"ValidateUnfreezeNft",
    );

    common_nft::ext(data.token_contract).nft_transfer(
        data.receiver_id,
        data.token_id,
        None,
        None,
    )
}
```

Listing 2.1: src/lib.rs

Similar problems also exist in functions `validate_transfer_nft()`, `withdraw_nft()`, and `freeze_nft()`.

Impact Users may lose their `NFT`s.

Suggestion I Implement corresponding logic to handle the potential failure of the cross-contract invocation in the callback function.

2.2 DeFi Security

2.2.1 Lack of Sanity Check in withdraw_nft()

Severity High

Status Fixed in `Version 3`

Introduced by `Version 1`

Description The function `withdraw_nft()` allows users to unfreeze `NFT`s in other chains after the corresponding `XPNFT`s are burned in `NEAR`'s mainnet. The contract will charge a certain fee for this operation. However, the amount of fees is decided by the user, and there is no check to guarantee that the fee is actually deposited into this contract.

The same problem also exists in the function `freeze_nft()`.

```
#[payable]
pub fn withdraw_nft(
    &mut self,
    token_contract: AccountId,
    token_id: TokenId,
    chain_nonce: u8,
    to: String,
    amt: U128,
) -> Promise {
    require!(!self.paused, "paused");

```

```
        require!(
            self.whitelist
                .contains_key(&token_contract.clone().to_string()),
            "Not whitelist"
        );

        xpnft::ext(token_contract.clone())
            .nft_token(token_id.clone())
            .then(Self::ext(env::current_account_id()).token_callback(
                token_contract,
                token_id,
                env::predecessor_account_id(),
                chain_nonce,
                to,
                amt.into(),
            ))
    }
```

Listing 2.2: src/lib.rs

```
    #[private]
    pub fn token_callback(
        &mut self,
        token_contract: AccountId,
        token_id: TokenId,
        owner_id: AccountId,
        chain_nonce: u8,
        to: String,
        amt: u128,
        #[callback_result] call_result: Result<Option<Token>, PromiseError>,
    ) -> Promise {
        require!(call_result.is_ok(), "token callback failed");

        xpnft::ext(token_contract.clone())
            .nft_burn(token_id.clone(), owner_id)
            .then(Self::ext(env::current_account_id()).withdraw_callback(
                token_contract,
                call_result.unwrap(),
                chain_nonce,
                to,
                amt.into(),
            ))
    }
```

Listing 2.3: src/lib.rs

```
    #[private]
    pub fn withdraw_callback(
        &mut self,
        token_contract: AccountId,
        token: Option<Token>,
        chain_nonce: u8,
        to: String,
        amt: u128,
        #[callback_result] call_result: Result<(), PromiseError>,
    ) -> Promise {
        require!(call_result.is_ok(), "withdraw failed");

        self.action_cnt += 1;
        self.tx_fees += amt;

        UnfreezeNftEvent {
            action_id: self.action_cnt,
            chain_nonce,
            to,
            amt,
            contract: token_contract,
            token,
        }
        .emit();

        Promise::new(env::current_account_id()).transfer(amt.into())
    }
```

Listing 2.4: src/lib.rs

```
    #[payable]
    pub fn freeze_nft(
        &mut self,
        token_contract: AccountId,
        token_id: TokenId,
        chain_nonce: u8,
        to: String,
        mint_with: String,
        amt: U128,
    ) -> Promise {
        require!(!self.paused, "paused");

        common_nft::ext(token_contract.clone())
            .with_attached_deposit(1)
            .nft_transfer(env::current_account_id(), token_id.clone(), None, None)
            .then(Self::ext(env::current_account_id()).freeze_callback(
                token_contract,
                token_id,
                chain_nonce,
                to,
                mint_with,
                amt.into(),
            ))
    }
```

Listing 2.5: src/lib.rs

```
    #[private]
    pub fn freeze_callback(
        &mut self,
        token_contract: AccountId,
        token_id: TokenId,
        chain_nonce: u8,
        to: String,
        mint_with: String,
        amt: u128,
        #[callback_result] call_result: Result<(), PromiseError>,
    ) -> Promise {
        require!(call_result.is_ok(), "freeze failed");

        self.action_cnt += 1;
        self.tx_fees += amt;

        TransferNftEvent {
            action_id: self.action_cnt,
            chain_nonce,
            to,
            amt,
            contract: token_contract,
            token_id,
            mint_with,
        }
        .emit();

        Promise::new(env::current_account_id()).transfer(amt.into())
    }
```

Listing 2.6: src/lib.rs

Impact Users can bridge their NFTs without fees. What's worse, the contract data XpBridge.tx_fees can be modified arbitrarily.

Suggestion I Add the check before updating the XpBridge.tx_fees to ensure the fees have been transferred to the contract and the fee amount is reasonable.

2.2.2 Lack of Specified Gas Distribution in Cross Contract Invocation

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description There is no check on whether the prepaid_gas is enough for function validate_transfer_nft(), withdraw_nft(), freeze_nft(), and validate_unfreeze_nft().

Specifically, since the gas distribution of the functions mentioned above is not specified, the default gas_weight will be set as 1. The remaining gas will be distributed equally to the invocation of the cross-contract call and the callback function.

```
    #[payable]
    pub fn validate_transfer_nft(
        &mut self,
        data: TransferNftData,
        sig_data: Vec<u8>,
) -> Promise {
    require!(!self.paused, "paused");

    self.require_sig(
        data.action_id.into(),
        data.try_to_vec().unwrap(),
        sig_data.into(),
        b"ValidateTransferNft",
    );

    xpnft::ext(data.mint_with)
        .with_attached_deposit(env::attached_deposit())
        .nft_mint(data.token_id, data.owner_id, data.token_metadata)
}
```

Listing 2.7: src/lib.rs

```
#[payable]
pub fn withdraw_nft(
    &mut self,
    token_contract: AccountId,
    token_id: TokenId,
    chain_nonce: u8,
    to: String,
    amt: U128,
) -> Promise {
    require!(!self.paused, "paused");

    require!(
        self.whitelist
            .contains_key(&token_contract.clone().to_string()),
        "Not whitelist"
    );

    xpnft::ext(token_contract.clone())
        .nft_token(token_id.clone())
        .then(Self::ext(env::current_account_id()).token_callback(
            token_contract,
            token_id,
            env::predecessor_account_id(),
            chain_nonce,
            to,
            amt.into(),
        ))
}
```

Listing 2.8: src/lib.rs

```
#[payable]
pub fn freeze_nft(
    &mut self,
    token_contract: AccountId,
    token_id: TokenId,
    chain_nonce: u8,
    to: String,
    mint_with: String,
    amt: U128,
) -> Promise {
    require!(!self.paused, "paused");

    common_nft::ext(token_contract.clone())
        .with_attached_deposit(1)
        .nft_transfer(env::current_account_id(), token_id.clone(), None, None)
        .then(Self::ext(env::current_account_id()).freeze_callback(
            token_contract,
            token_id,
            chain_nonce,
            to,
            mint_with,
            amt.into(),
        ))
}
```

Listing 2.9: src/lib.rs

```
    #[payable]
    pub fn validate_unfreeze_nft(
        &mut self,
        data: UnfreezeNftData,
        sig_data: Vec<u8>,
    ) -> Promise {
        require!(!self.paused, "paused");

        self.require_sig(
            data.action_id.into(),
            data.try_to_vec().unwrap(),
            sig_data,
            b"ValidateUnfreezeNft",
        );

        common_nft::ext(data.token_contract).nft_transfer(
            data.receiver_id,
            data.token_id,
            None,
            None,
        )
    }
```

Listing 2.10: src/lib.rs

Impact The callback functions will fail if there is not enough gas left. This can result in the incorrect contract state.

Suggestion I Specify the gas prepared for cross-contract invocations and callback functions.

2.2.3 Improper Calculation for Storage Balance

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description Function validate_withdraw_fees() allows the contract to withdraw NFT transaction fees. The design aims to leave a certain amount of NEARs for the storage staking. However, the available withdrawn NEAR is calculated with self.tx_fees - storage_used, which is incorrect. This is because the self.tx_fees does not equal to near_sdk::env::account_balance(), which represents the amount of NEARs remaining in the contract.

```rust
#[payable]
 pub fn validate_withdraw_fees(
     &mut self,
     data: WithdrawFeeData,
     sig_data: Vec<u8>,
 ) -> Promise {
     require!(!self.paused, "paused");

     self.require_sig(
         data.action_id.into(),
         data.try_to_vec().unwrap(),
         sig_data,
         b"WithdrawFees",
     );

     let storage_used = env::storage_usage();
     let amt = self.tx_fees - storage_used as u128 * env::storage_byte_cost();
     Promise::new(env::current_account_id())
         .transfer(amt)
         .then(Self::ext(env::current_account_id()).withdraw_fee_callback())
 }
```

Listing 2.11: src/lib.rs

```rust
#[private]
 pub fn withdraw_fee_callback(
     &mut self,
     #[callback_result] call_result: Result<(), PromiseError>,
 ) {
     require!(call_result.is_ok(), "withdraw failed");

     self.tx_fees = 0;
 }
```

Listing 2.12: src/lib.rs

Impact The current calculation is against the design purpose of the contract.

Suggestion I Add the check to ensure the amount of NEARs withdrawn from the tx_fees is less than the total amount of NEARs (near_sdk::env::account_balance()) in the contract. Meanwhile, the logic in the callback function should be revised accordingly.

2.2.4 Lack of Whitelist Check in freeze_nft()

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description Function freeze_nft() allows the user to freeze the NFT on the NEAR's mainnet to get a minted XPNFT token in another chain. However, there is no whitelist check for the input NFT token contracts, which means any NFT tokens can be bridged to other chains.

```
#[payable]
 pub fn freeze_nft(
     &mut self,
     token_contract: AccountId,
     token_id: TokenId,
     chain_nonce: u8,
     to: String,
     mint_with: String,
     amt: U128,
 ) -> Promise {
     require!(!self.paused, "paused");

     common_nft::ext(token_contract.clone())
         .with_attached_deposit(1)
         .nft_transfer(env::current_account_id(), token_id.clone(), None, None)
         .then(Self::ext(env::current_account_id()).freeze_callback(
             token_contract,
             token_id,
             chain_nonce,
             to,
             mint_with,
             amt.into(),
         ))
 }
```

Listing 2.13: src/lib.rs

Impact Any NFT tokens can be bridged to other chains, which is a waste of gas for the relayers.

Suggestion I Add the whitelist check in function freeze_nft() to ensure the users' NFTs are valuable.

2.2.5 Incorrect tx_fee Receiver

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description The callback function withdraw_callback() will transfer tx_fees to the contract itself (line 378), which is meaningless. The same problem also exists in function freeze_callback() and function validate_withdraw_fees().

```
353     #[private]
```

```rust
pub fn withdraw_callback(
    &mut self,
    token_contract: AccountId,
    token: Option<Token>,
    chain_nonce: u8,
    to: String,
    amt: u128,
    #[callback_result] call_result: Result<(), PromiseError>,
) -> Promise {
    require!(call_result.is_ok(), "withdraw failed");

    self.action_cnt += 1;
    self.tx_fees += amt;

    UnfreezeNftEvent {
        action_id: self.action_cnt,
        chain_nonce,
        to,
        amt,
        contract: token_contract,
        token,
    }
    .emit();

    Promise::new(env::current_account_id()).transfer(amt.into())
}
```

Listing 2.14: src/lib.rs

```rust
#[payable]
pub fn validate_withdraw_fees(
    &mut self,
    data: WithdrawFeeData,
    sig_data: Vec<u8>,
) -> Promise {
    require!(!self.paused, "paused");

    self.require_sig(
        data.action_id.into(),
        data.try_to_vec().unwrap(),
        sig_data,
        b"WithdrawFees",
    );

    let storage_used = env::storage_usage();
    let amt = self.tx_fees - storage_used as u128 * env::storage_byte_cost();
    Promise::new(env::current_account_id())
        .transfer(amt)
        .then(Self::ext(env::current_account_id()).withdraw_fee_callback())
}
```

Listing 2.15: src/lib.rs

```
#[private]
pub fn freeze_callback(
    &mut self,
    token_contract: AccountId,
    token_id: TokenId,
    chain_nonce: u8,
    to: String,
    mint_with: String,
    amt: u128,
    #[callback_result] call_result: Result<(), PromiseError>,
) -> Promise {
    require!(call_result.is_ok(), "freeze failed");

    self.action_cnt += 1;
    self.tx_fees += amt;

    TransferNftEvent {
        action_id: self.action_cnt,
        chain_nonce,
        to,
        amt,
        contract: token_contract,
        token_id,
        mint_with,
    }
    .emit();

    Promise::new(env::current_account_id()).transfer(amt.into())
}
```

Listing 2.16: src/lib.rs

Impact The tx_fees will be locked in the contract.

Suggestion I Either transfer tx_fees to a reasonable receiver or remove the redundant transfer in the mentioned functions above.

2.3 Additional Recommendation

2.3.1 Redundant Usage of Macro payable

Status Fixed in Version 2

Introduced by Version 1

Description Payable methods are able to accept token transfers together with function calls. However, a few functions in the contract actually do not need this functionality. They are validate_blacklist(), validate_whitelist(), validate_update_group_key(), validate_withdraw_fees(), validate_unpause(), validate_pause(), withdraw_nft(), and validate_unfreeze_nft().

```
#[payable]
pub fn remove_guardians(&mut self, guardians: Vec<ValidAccountId>) {
```

```rust
        assert_one_yocto();
        self.assert_owner();
        for guardian in guardians {
            self.guardians.remove(guardian.as_ref());
        }
    }
```

Listing 2.17: src/lib.rs

```rust
    #[payable]
    pub fn validate_unpause(&mut self, data: UnpauseData, sig_data: Vec<u8>) {
        require!(self.paused, "unpaused");

        self.require_sig(
            data.action_id.into(),
            data.try_to_vec().unwrap(),
            sig_data,
            b"SetUnpause",
        );

        self.paused = false;
    }
```

Listing 2.18: src/lib.rs

```rust
    #[payable]
    pub fn validate_withdraw_fees(
        &mut self,
        data: WithdrawFeeData,
        sig_data: Vec<u8>,
    ) -> Promise {
        require!(!self.paused, "paused");

        self.require_sig(
            data.action_id.into(),
            data.try_to_vec().unwrap(),
            sig_data,
            b"WithdrawFees",
        );

        let storage_used = env::storage_usage();
        let amt = self.tx_fees - storage_used as u128 * env::storage_byte_cost();
        Promise::new(env::current_account_id())
            .transfer(amt)
            .then(Self::ext(env::current_account_id()).withdraw_fee_callback())
    }
```

Listing 2.19: src/lib.rs

```rust
    #[payable]
    pub fn validate_update_group_key(&mut self, data: UpdateGroupkeyData, sig_data: Vec<u8>) {
        require!(!self.paused, "paused");

        self.require_sig(
            data.action_id.into(),
            data.try_to_vec().unwrap(),
            sig_data,
            b"SetGroupKey",
        );

        self.group_key = data.group_key;
    }
```

Listing 2.20: src/lib.rs

```rust
    #[payable]
    pub fn validate_whitelist(&mut self, data: WhitelistData, sig_data: Base64VecU8) {
        require!(!self.paused, "paused");

        require!(
            !self
                .whitelist
                .contains_key(&data.token_contract.to_string()),
            "Already whitelist"
        );

        self.require_sig(
            data.action_id.into(),
            data.try_to_vec().unwrap(),
            sig_data.into(),
            b"WhitelistNft",
        );

        self.whitelist.insert(data.token_contract, true);
    }
```

Listing 2.21: src/lib.rs

```rust
    #[payable]
    pub fn validate_blacklist(&mut self, data: WhitelistData, sig_data: Vec<u8>) {
        require!(!self.paused, "paused");

        require!(
            self.whitelist
                .contains_key(&data.token_contract.to_string()),
            "Not whitelist"
        );

        self.require_sig(
            data.action_id.into(),
            data.try_to_vec().unwrap(),
            sig_data,
            b"ValidateBlacklistNft"
        );

        self.whitelist.remove(&data.token_contract);
```

```
}
```

Listing 2.22: src/lib.rs

```
#[payable]
pub fn withdraw_nft(
    &mut self,
    token_contract: AccountId,
    token_id: TokenId,
    chain_nonce: u8,
    to: String,
    amt: U128,
) -> Promise {
    require!(!self.paused, "paused");

    require!(
        self.whitelist
            .contains_key(&token_contract.clone().to_string()),
        "Not whitelist"
    );

    xpnft::ext(token_contract.clone())
        .nft_token(token_id.clone())
        .then(Self::ext(env::current_account_id()).token_callback(
            token_contract,
            token_id,
            env::predecessor_account_id(),
            chain_nonce,
            to,
            amt.into(),
        ))
}
```

Listing 2.23: src/lib.rs

```
#[payable]
pub fn validate_unfreeze_nft(
    &mut self,
    data: UnfreezeNftData,
    sig_data: Vec<u8>,
) -> Promise {
    require!(!self.paused, "paused");

    self.require_sig(
        data.action_id.into(),
        data.try_to_vec().unwrap(),
        sig_data,
        b"ValidateUnfreezeNft",
    );

    common_nft::ext(data.token_contract).nft_transfer(
        data.receiver_id,
        data.token_id,
        None,
        None,
    )
}
```

Listing 2.24: src/lib.rs

Suggestion I Remove the macro `#[payable]` in function `validate_blacklist()`, `validate_whitelist()`, `validate_update_group_key()`, `validate_withdraw_fees()`, `validate_unpause()`, `validate_pause()`, and `validate_unfreeze_nft()`. Add internal function `assert_one_yocto()` in function `withdraw_nft()`.

2.3.2 Improper Usage of the Data Structures

Status Fixed in `Version 3`

Introduced by `Version 1`

Description The data structure `HashMap` is implemented for storing the related data of `consumed_actions` and `whitelist`. However, the value in these two structures is never checked or used. For example, when removing members from the `whitelist` via function `validate_blacklist()`, it will check whether the target is in the `whitelist` first. As shown below, only the keys of the `whitelist` are checked, and the value of the corresponding key will never be checked (lines 249 - 252).

```
#[near_bindgen]
#[derive(Default, BorshDeserialize, BorshSerialize)]
pub struct XpBridge {
    consumed_actions: HashMap<u128, bool>,
    paused: bool,
    tx_fees: u128,
    group_key: [u8; 32],
    action_cnt: u128,
    whitelist: HashMap<String, bool>,
}
```

Listing 2.25: src/lib.rs

```
244    #[payable]
245    pub fn validate_blacklist(&mut self, data: WhitelistData, sig_data: Vec<u8>) {
246        require!(!self.paused, "paused");
247
248        require!(
249            self.whitelist
250                .contains_key(&data.token_contract.to_string()),
251            "Not whitelist"
252        );
253
254        self.require_sig(
255            data.action_id.into(),
256            data.try_to_vec().unwrap(),
257            sig_data,
258            b"ValidateBlacklistNft"
259        );
260
```

```
        self.whitelist.remove(&data.token_contract);
    }
```

Listing 2.26: src/lib.rs

Suggestion I It's recommended to use `UnorderedSet` for `consumed_actions` and `whitelist`.

2.3.3 Inconsistent Type of Input in validate_whitelist()

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The parameter `sig_data` input in the function `validate_whitelist()` is `Base64VecU8`. However, the type of `sig_data` input in the other functions of the contract is `Vec<u8>`, which is inconsistent.

```
#[payable]
    pub fn validate_whitelist(&mut self, data: WhitelistData, sig_data: Base64VecU8) {
        require!(!self.paused, "paused");

        require!(
            !self
                .whitelist
                .contains_key(&data.token_contract.to_string()),
            "Already whitelist"
        );

        self.require_sig(
            data.action_id.into(),
            data.try_to_vec().unwrap(),
            sig_data.into(),
            b"WhitelistNft",
        );

        self.whitelist.insert(data.token_contract, true);
    }
```

Listing 2.27: src/lib.rs

Suggestion I It is recommended to change the type of parameter `sig_data` in the function `validate_whitelist()` to `Vec<u8>`.

2.3.4 Improper Use of Private Function

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Macro `#[private]` is usually used for the callback function of a cross-contract call as the callback function should only be called by the contract itself. However, the function `require_sig()` is not a callback function and is currently implemented with the macro `#[private]`, which is incorrect.

In the current implementation, the function acts as a helper function (internal function), which is invoked by the privileged public functions for the purpose of signature check.

```
#[private]
fn require_sig(&mut self, action_id: u128, data: Vec<u8>, sig_data: Vec<u8>, context: &[u8]) {
    let f = self.consumed_actions.contains_key(&action_id);
    require!(!f, "Duplicated Action");

    self.consumed_actions.insert(action_id, true);

    let mut hasher = Sha512::new();
    hasher.update(context);
    hasher.update(data);
    let hash = hasher.finalize();

    let sig = Signature::new(sig_data.as_slice().try_into().unwrap());
    let key = PublicKey::new(self.group_key);
    let res = key.verify(hash, &sig);
    require!(res.is_ok(), "Unauthorized Action");
}
```

Listing 2.28: src/lib.rs

Suggestion I It is recommended to remove the macro `#[private]` for function `require_sig()`.

2.3.5 Potential Centralization Problem

Status Confirmed

Introduced by `Version 1`

Description This project has potential centralization problems. The off-chain relayer has the privilege to configure several system parameters (e.g., `group_key`), mint `XPNFT` tokens, withdraw freezed `NFT` tokens of users, and pause/unpause the contract. Besides, the person who has the private key of the contract can be able to transfer operation fees and freezed `NFT`s of users.

Suggestion I A decentralization design for the off-chain signature verification is recommended. Also, it's suggested to delete the private key of the contract, and implement an upgrade function for the further maintenance.

Feedback from the Project This is taken care of by the `BFT` multisignature of the validators. Every such change requires the consensus of `2/3+1`.

2.3.6 Lack of Check of Updated group_key

Status Confirmed

Introduced by `Version 1`

Description The system parameter `group_key` can be updated via the function `validate_update_group_key()`. However, the validation of the updated `group_key` is not implemented. In this case, when an incorrect `group_key` is provided, the contract is under the risk of attack and the privileged function can not be invoked.

```
#[payable]
pub fn validate_update_group_key(&mut self, data: UpdateGroupkeyData, sig_data: Vec<u8>) {
   require!(!self.paused, "paused");

   self.require_sig(
       data.action_id.into(),
       data.try_to_vec().unwrap(),
       sig_data,
       b"SetGroupKey",
   );

   self.group_key = data.group_key;
}
```

Listing 2.29: src/lib.rs

Suggestion I It's suggested to validate the updated `group_key` using the corresponding data signed with the new private key.

Feedback from the Project The `FROST Group Key` is generated by an algorithm based on the set threshold of signers, their total number and takes into account their private and public keys. The contract has no way of checking this key's validity from within.

2.4 Notes

2.4.1 Secure Implementation of the Offchain Mechanism

Status Confirmed

Introduced by `Version 1`

Description The protocol depends on an off-chain mechanism that monitors and validates the on-chain data before bridging `NFT`s across multiple chains, which is not included in our audit scope. In this case, we assume the validators would properly verify the data before signing the related data.

2.4.2 Fee Amount Calculation

Status Confirmed

Introduced by `Version 1`

Description The protocol charges a certain amount of operation fees from users for bridging their `NFT`s among different chains. The algorithm for calculating the operation fee is out of the audit scope. Meanwhile, if the user sends more fees than the estimated value, the additional amount will not be refunded, which should be noted.