



BlockSec

Security Audit Report for XP-Near-Integration

Date: Jan 12, 2023

Version: 2.0

Contact: contact@blocksec.com

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

About BlockSec The **BlockSec Team** focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team 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 released detailed analysis reports of high-impact security incidents. 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 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

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.

²https://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 <code>withdraw_nft()</code>	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 <code>freeze_nft()</code>	DeFi Security	Fixed
6	High	Incorrect <code>tx_fee</code> 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 <code>validate_whitelist()</code>	Recommendation	Fixed
10	-	Improper Use of Private Function	Recommendation	Fixed
11	-	Potential Centralization Problem	Recommendation	Confirmed
12	-	Lack of Check of Updated <code>group_key</code>	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.

```
446     #[payable]
447     pub fn validate_unfreeze_nft(
448         &mut self,
449         data: UnfreezeNftData,
450         sig_data: Vec<u8>,
```

```
451 ) -> Promise {
452     require!(!self.paused, "paused");
453
454     self.require_sig(
455         data.action_id.into(),
456         data.try_to_vec().unwrap(),
457         sig_data,
458         b"ValidateUnfreezeNft",
459     );
460
461     common_nft::ext(data.token_contract).nft_transfer(
462         data.receiver_id,
463         data.token_id,
464         None,
465         None,
466     )
467 }
```

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 `NFTs`.

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 `NFTs` in other chains after the corresponding `XPNTs` 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()`.

```
293 #[payable]
294 pub fn withdraw_nft(
295     &mut self,
296     token_contract: AccountId,
297     token_id: TokenId,
298     chain_nonce: u8,
299     to: String,
300     amt: U128,
301 ) -> Promise {
302     require!(!self.paused, "paused");
303 }
```



```
304     require!(
305         self.whitelist
306         .contains_key(&token_contract.clone().to_string()),
307         "Not whitelist"
308     );
309
310     xpnft::ext(token_contract.clone())
311         .nft_token(token_id.clone())
312         .then(Self::ext(env::current_account_id()).token_callback(
313             token_contract,
314             token_id,
315             env::predecessor_account_id(),
316             chain_nonce,
317             to,
318             amt.into(),
319         ))
320 }
```

Listing 2.2: src/lib.rs

```
326     #[private]
327     pub fn token_callback(
328         &mut self,
329         token_contract: AccountId,
330         token_id: TokenId,
331         owner_id: AccountId,
332         chain_nonce: u8,
333         to: String,
334         amt: u128,
335         #[callback_result] call_result: Result<Option<Token>, PromiseError>,
336     ) -> Promise {
337         require!(call_result.is_ok(), "token callback failed");
338
339         xpnft::ext(token_contract.clone())
340             .nft_burn(token_id.clone(), owner_id)
341             .then(Self::ext(env::current_account_id()).withdraw_callback(
342                 token_contract,
343                 call_result.unwrap(),
344                 chain_nonce,
345                 to,
346                 amt.into(),
347             ))
348     }
```

Listing 2.3: src/lib.rs

```
353     #[private]
354     pub fn withdraw_callback(
355         &mut self,
356         token_contract: AccountId,
357         token: Option<Token>,
358         chain_nonce: u8,
359         to: String,
```

```
360     amt: u128,  
361     #[callback_result] call_result: Result<(), PromiseError>,  
362 ) -> Promise {  
363     require!(call_result.is_ok(), "withdraw failed");  
364  
365     self.action_cnt += 1;  
366     self.tx_fees += amt;  
367  
368     UnfreezeNftEvent {  
369         action_id: self.action_cnt,  
370         chain_nonce,  
371         to,  
372         amt,  
373         contract: token_contract,  
374         token,  
375     }  
376     .emit();  
377  
378     Promise::new(env::current_account_id()).transfer(amt.into())  
379 }
```

Listing 2.4: src/lib.rs

```
384     #[payable]  
385     pub fn freeze_nft(  
386         &mut self,  
387         token_contract: AccountId,  
388         token_id: TokenId,  
389         chain_nonce: u8,  
390         to: String,  
391         mint_with: String,  
392         amt: U128,  
393     ) -> Promise {  
394         require!(!self.paused, "paused");  
395  
396         common_nft::ext(token_contract.clone())  
397             .with_attached_deposit(1)  
398             .nft_transfer(env::current_account_id(), token_id.clone(), None, None)  
399             .then(Self::ext(env::current_account_id()).freeze_callback(  
400                 token_contract,  
401                 token_id,  
402                 chain_nonce,  
403                 to,  
404                 mint_with,  
405                 amt.into(),  
406             ))  
407     }
```

Listing 2.5: src/lib.rs

```
413     #[private]  
414     pub fn freeze_callback(  
415         &mut self,
```

```
416     token_contract: AccountId,
417     token_id: TokenId,
418     chain_nonce: u8,
419     to: String,
420     mint_with: String,
421     amt: u128,
422     #[callback_result] call_result: Result<(), PromiseError>,
423 ) -> Promise {
424     require!(call_result.is_ok(), "freeze failed");
425
426     self.action_cnt += 1;
427     self.tx_fees += amt;
428
429     TransferNftEvent {
430         action_id: self.action_cnt,
431         chain_nonce,
432         to,
433         amt,
434         contract: token_contract,
435         token_id,
436         mint_with,
437     }
438     .emit();
439
440     Promise::new(env::current_account_id()).transfer(amt.into())
441 }
```

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.

```
268     #[payable]
269     pub fn validate_transfer_nft(
270         &mut self,
271         data: TransferNftData,
272         sig_data: Vec<u8>,
```

```
273 ) -> Promise {
274     require!(!self.paused, "paused");
275
276     self.require_sig(
277         data.action_id.into(),
278         data.try_to_vec().unwrap(),
279         sig_data.into(),
280         b"ValidateTransferNft",
281     );
282
283     xpnft::ext(data.mint_with)
284         .with_attached_deposit(env::attached_deposit())
285         .nft_mint(data.token_id, data.owner_id, data.token_metadata)
286 }
```

Listing 2.7: src/lib.rs

```
293 #[payable]
294 pub fn withdraw_nft(
295     &mut self,
296     token_contract: AccountId,
297     token_id: TokenId,
298     chain_nonce: u8,
299     to: String,
300     amt: U128,
301 ) -> Promise {
302     require!(!self.paused, "paused");
303
304     require!(
305         self.whitelist
306             .contains_key(&token_contract.clone().to_string()),
307         "Not whitelist"
308     );
309
310     xpnft::ext(token_contract.clone())
311         .nft_token(token_id.clone())
312         .then(Self::ext(env::current_account_id()).token_callback(
313             token_contract,
314             token_id,
315             env::predecessor_account_id(),
316             chain_nonce,
317             to,
318             amt.into(),
319         ))
320 }
```

Listing 2.8: src/lib.rs

```
384 #[payable]
385 pub fn freeze_nft(
386     &mut self,
387     token_contract: AccountId,
388     token_id: TokenId,
```

```
389     chain_nonce: u8,  
390     to: String,  
391     mint_with: String,  
392     amt: U128,  
393 ) -> Promise {  
394     require!(!self.paused, "paused");  
395  
396     common_nft::ext(token_contract.clone())  
397         .with_attached_deposit(1)  
398         .nft_transfer(env::current_account_id(), token_id.clone(), None, None)  
399         .then(Self::ext(env::current_account_id()).freeze_callback(  
400             token_contract,  
401             token_id,  
402             chain_nonce,  
403             to,  
404             mint_with,  
405             amt.into(),  
406         ))  
407 }
```

Listing 2.9: src/lib.rs

```
446 #[payable]  
447 pub fn validate_unfreeze_nft(  
448     &mut self,  
449     data: UnfreezeNftData,  
450     sig_data: Vec<u8>,  
451 ) -> Promise {  
452     require!(!self.paused, "paused");  
453  
454     self.require_sig(  
455         data.action_id.into(),  
456         data.try_to_vec().unwrap(),  
457         sig_data,  
458         b"ValidateUnfreezeNft",  
459     );  
460  
461     common_nft::ext(data.token_contract).nft_transfer(  
462         data.receiver_id,  
463         data.token_id,  
464         None,  
465         None,  
466     )  
467 }
```

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.

```
161  #[payable]
162  pub fn validate_withdraw_fees(
163      &mut self,
164      data: WithdrawFeeData,
165      sig_data: Vec<u8>,
166  ) -> Promise {
167      require!(!self.paused, "paused");
168
169      self.require_sig(
170          data.action_id.into(),
171          data.try_to_vec().unwrap(),
172          sig_data,
173          b"WithdrawFees",
174      );
175
176      let storage_used = env::storage_usage();
177      let amt = self.tx_fees - storage_used as u128 * env::storage_byte_cost();
178      Promise::new(env::current_account_id())
179          .transfer(amt)
180          .then(Self::ext(env::current_account_id()).withdraw_fee_callback())
181  }
```

Listing 2.11: `src/lib.rs`

```
186  #[private]
187  pub fn withdraw_fee_callback(
188      &mut self,
189      #[callback_result] call_result: Result<(), PromiseError>,
190  ) {
191      require!(call_result.is_ok(), "withdraw failed");
192
193      self.tx_fees = 0;
194  }
```

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.

```
384  #[payable]
385  pub fn freeze_nft(
386      &mut self,
387      token_contract: AccountId,
388      token_id: TokenId,
389      chain_nonce: u8,
390      to: String,
391      mint_with: String,
392      amt: U128,
393  ) -> Promise {
394      require(!self.paused, "paused");
395
396      common_nft::ext(token_contract.clone())
397          .with_attached_deposit(1)
398          .nft_transfer(env::current_account_id(), token_id.clone(), None, None)
399          .then(Self::ext(env::current_account_id()).freeze_callback(
400              token_contract,
401              token_id,
402              chain_nonce,
403              to,
404              mint_with,
405              amt.into(),
406          ))
407  }
```

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' [NFT](#)s 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]
```

```
354 pub fn withdraw_callback(
355     &mut self,
356     token_contract: AccountId,
357     token: Option<Token>,
358     chain_nonce: u8,
359     to: String,
360     amt: u128,
361     #[callback_result] call_result: Result<(), PromiseError>,
362 ) -> Promise {
363     require!(call_result.is_ok(), "withdraw failed");
364
365     self.action_cnt += 1;
366     self.tx_fees += amt;
367
368     UnfreezeNftEvent {
369         action_id: self.action_cnt,
370         chain_nonce,
371         to,
372         amt,
373         contract: token_contract,
374         token,
375     }
376     .emit();
377
378     Promise::new(env::current_account_id()).transfer(amt.into())
379 }
```

Listing 2.14: src/lib.rs

```
161 #[payable]
162 pub fn validate_withdraw_fees(
163     &mut self,
164     data: WithdrawFeeData,
165     sig_data: Vec<u8>,
166 ) -> Promise {
167     require!(!self.paused, "paused");
168
169     self.require_sig(
170         data.action_id.into(),
171         data.try_to_vec().unwrap(),
172         sig_data,
173         b"WithdrawFees",
174     );
175
176     let storage_used = env::storage_usage();
177     let amt = self.tx_fees - storage_used as u128 * env::storage_byte_cost();
178     Promise::new(env::current_account_id())
179         .transfer(amt)
180         .then(Self::ext(env::current_account_id()).withdraw_fee_callback())
181 }
```

Listing 2.15: src/lib.rs


```
413     #[private]
414     pub fn freeze_callback(
415         &mut self,
416         token_contract: AccountId,
417         token_id: TokenId,
418         chain_nonce: u8,
419         to: String,
420         mint_with: String,
421         amt: u128,
422         #[callback_result] call_result: Result<(), PromiseError>,
423     ) -> Promise {
424         require!(call_result.is_ok(), "freeze failed");
425
426         self.action_cnt += 1;
427         self.tx_fees += amt;
428
429         TransferNftEvent {
430             action_id: self.action_cnt,
431             chain_nonce,
432             to,
433             amt,
434             contract: token_contract,
435             token_id,
436             mint_with,
437         }
438         .emit();
439
440         Promise::new(env::current_account_id()).transfer(amt.into())
441     }
```

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()`.

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

```
128     assert_one_yocto();
129     self.assert_owner();
130     for guardian in guardians {
131         self.guardians.remove(guardian.as_ref());
132     }
133 }
```

Listing 2.17: src/lib.rs

```
143     #[payable]
144     pub fn validate_unpause(&mut self, data: UnpauseData, sig_data: Vec<u8>) {
145         require!(self.paused, "unpaused");
146
147         self.require_sig(
148             data.action_id.into(),
149             data.try_to_vec().unwrap(),
150             sig_data,
151             b"SetUnpause",
152         );
153
154         self.paused = false;
155     }
```

Listing 2.18: src/lib.rs

```
161     #[payable]
162     pub fn validate_withdraw_fees(
163         &mut self,
164         data: WithdrawFeeData,
165         sig_data: Vec<u8>,
166     ) -> Promise {
167         require!(!self.paused, "paused");
168
169         self.require_sig(
170             data.action_id.into(),
171             data.try_to_vec().unwrap(),
172             sig_data,
173             b"WithdrawFees",
174         );
175
176         let storage_used = env::storage_usage();
177         let amt = self.tx_fees - storage_used as u128 * env::storage_byte_cost();
178         Promise::new(env::current_account_id())
179             .transfer(amt)
180             .then(Self::ext(env::current_account_id()).withdraw_fee_callback())
181     }
```

Listing 2.19: src/lib.rs

```
198     #[payable]
199     pub fn validate_update_group_key(&mut self, data: UpdateGroupkeyData, sig_data: Vec<u8>) {
200         require!(!self.paused, "paused");
201     }
```

```
202     self.require_sig(  
203         data.action_id.into(),  
204         data.try_to_vec().unwrap(),  
205         sig_data,  
206         b"SetGroupKey",  
207     );  
208  
209     self.group_key = data.group_key;  
210 }
```

Listing 2.20: src/lib.rs

```
218     #[payable]  
219     pub fn validate_whitelist(&mut self, data: WhitelistData, sig_data: Base64VecU8) {  
220         require!(!self.paused, "paused");  
221  
222         require!(  
223             !self  
224                 .whitelist  
225                 .contains_key(&data.token_contract.to_string()),  
226             "Already whitelist"  
227         );  
228  
229         self.require_sig(  
230             data.action_id.into(),  
231             data.try_to_vec().unwrap(),  
232             sig_data.into(),  
233             b"WhitelistNft",  
234         );  
235  
236         self.whitelist.insert(data.token_contract, true);  
237     }
```

Listing 2.21: 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  
261         self.whitelist.remove(&data.token_contract);
```

```
262 }
```

Listing 2.22: src/lib.rs

```
293 #[payable]
294 pub fn withdraw_nft(
295     &mut self,
296     token_contract: AccountId,
297     token_id: TokenId,
298     chain_nonce: u8,
299     to: String,
300     amt: U128,
301 ) -> Promise {
302     require!(!self.paused, "paused");
303
304     require!(
305         self.whitelist
306             .contains_key(&token_contract.clone().to_string()),
307         "Not whitelist"
308     );
309
310     xpnft::ext(token_contract.clone())
311         .nft_token(token_id.clone())
312         .then(Self::ext(env::current_account_id()).token_callback(
313             token_contract,
314             token_id,
315             env::predecessor_account_id(),
316             chain_nonce,
317             to,
318             amt.into(),
319         ))
320 }
```

Listing 2.23: src/lib.rs

```
446 #[payable]
447 pub fn validate_unfreeze_nft(
448     &mut self,
449     data: UnfreezeNftData,
450     sig_data: Vec<u8>,
451 ) -> Promise {
452     require!(!self.paused, "paused");
453
454     self.require_sig(
455         data.action_id.into(),
456         data.try_to_vec().unwrap(),
457         sig_data,
458         b"ValidateUnfreezeNft",
459     );
460
461     common_nft::ext(data.token_contract).nft_transfer(
462         data.receiver_id,
463         data.token_id,
```

```
464         None,
465         None,
466     )
467 }
```

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).

```
69 #[near_bindgen]
70 #[derive(Default, BorshDeserialize, BorshSerialize)]
71 pub struct XpBridge {
72     consumed_actions: HashMap<u128, bool>,
73     paused: bool,
74     tx_fees: u128,
75     group_key: [u8; 32],
76     action_cnt: u128,
77     whitelist: HashMap<String, bool>,
78 }
```

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     }
```

```
261         self.whitelist.remove(&data.token_contract);
262     }
```

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.

```
218 #[payable]
219 pub fn validate_whitelist(&mut self, data: WhitelistData, sig_data: Base64VecU8) {
220     require!(!self.paused, "paused");
221
222     require!(
223         !self
224             .whitelist
225             .contains_key(&data.token_contract.to_string()),
226         "Already whitelist"
227     );
228
229     self.require_sig(
230         data.action_id.into(),
231         data.try_to_vec().unwrap(),
232         sig_data.into(),
233         b"WhitelistNft",
234     );
235
236     self.whitelist.insert(data.token_contract, true);
237 }
```

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.

```
105  #[private]
106  fn require_sig(&mut self, action_id: u128, data: Vec<u8>, sig_data: Vec<u8>, context: &[u8]) {
107      let f = self.consumed_actions.contains_key(&action_id);
108      require!(!f, "Duplicated Action");
109
110      self.consumed_actions.insert(action_id, true);
111
112      let mut hasher = Sha512::new();
113      hasher.update(context);
114      hasher.update(data);
115      let hash = hasher.finalize();
116
117      let sig = Signature::new(sig_data.as_slice().try_into().unwrap());
118      let key = PublicKey::new(self.group_key);
119      let res = key.verify(hash, &sig);
120      require!(res.is_ok(), "Unauthorized Action");
121  }
```

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 relay has the privilege to configure several system parameters (e.g., `group_key`), mint `XP NFT` 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 `NFTs` 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.

```
198  #[payable]
199  pub fn validate_update_group_key(&mut self, data: UpdateGroupkeyData, sig_data: Vec<u8>) {
```

```
200     require(!self.paused, "paused");
201
202     self.require_sig(
203         data.action_id.into(),
204         data.try_to_vec().unwrap(),
205         sig_data,
206         b"SetGroupKey",
207     );
208
209     self.group_key = data.group_key;
210 }
```

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 `NFTs` 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 `NFTs` 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.