

Security Audit

Report for Allstake Near Contract

Date: July 05, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction	1
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
Chapter 2 Findings	4
2.1 DeFi Security	4
2.1.1 Incomplete state reversion in function <code>resolve_complete_queued_withdrawals()</code>	4
2.1.2 Lack of storage fee check	8
2.1.3 Potential insufficient storage fee	11
2.1.4 Lack of FT transfer fee charge	11
2.1.5 Lack of check in privileged functions	13
2.2 Additional Recommendation	14
2.2.1 Lack of check in function <code>storage_deposit()</code>	14
2.2.2 Redundant code	15
2.2.3 Lack of check in function <code>add_strategies_to_deposit_whitelist()</code> and <code>remove_strategies_from_deposit_whitelist()</code>	16
2.2.4 Lack of check in function <code>set_strategies_withdrawal_delay()</code>	17
2.3 Note	18
2.3.1 Potential centralization risk	18

Report Manifest

Item	Description
Client	Allstake
Target	Allstake Near Contract

Version History

Version	Date	Description
1.0	July 05, 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 target of this audit is the code repository of Allstake Near Contract¹ of Allstake.

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
Allstake Near Contract	Version 1	c1064fcd50815080d8d9874a427cdc5772a9607e
	Version 2	985adc34881688f5a737044b630cb42db5c46f21

1.2 Disclaimer

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

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

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

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

¹<https://github.com/allstake/allstake>

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

1.3.1 Software Security

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

1.3.2 DeFi Security

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

1.3.3 NFT Security

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

1.3.4 Additional Recommendation

- * Gas optimization

* Code quality and style



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

1.4 Security Model

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

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

Table 1.1: Vulnerability Severity Classification

Impact	<i>High</i>	High	Medium
	<i>Low</i>	Medium	Low
		<i>High</i>	<i>Low</i>
		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 found **five** potential security issues. Besides, we have **four** recommendations and **one** note.

- Medium Risk: 3
- Low Risk: 2
- Recommendation: 4
- Note: 1

ID	Severity	Description	Category	Status
1	Medium	Incomplete state reversion in function <code>resolve_complete_queued_withdrawals()</code>	DeFi Security	Fixed
2	Medium	Lack of storage fee check	DeFi Security	Fixed
3	Medium	Potential insufficient storage fee	DeFi Security	Fixed
4	Low	Lack of FT transfer fee charge	DeFi Security	Confirmed
5	Low	Lack of check in privileged functions	DeFi Security	Fixed
6	-	Lack of check in function <code>storage_deposit()</code>	Recommendation	Fixed
7	-	Redundant code	Recommendation	Fixed
8	-	Lack of check in function <code>add_strategies_to_deposit_whitelist()</code> and <code>remove_strategies_from_deposit_whitelist()</code>	Recommendation	Fixed
9	-	Lack of check in function <code>set_strategies_withdrawal_delay()</code>	Recommendation	Fixed
10	-	Potential centralization risk	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Incomplete state reversion in function

`resolve_complete_queued_withdrawals()`

Severity Medium

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description In the file `delegation.rs`, users can retrieve their deposited assets through the function `complete_queued_withdrawals()`. From lines 522-531, if the transfer fails, the contract's function `resolve_complete_queued_withdrawals()` is invoked to roll back the user's state. Specifically, when a user successfully withdraws, the function `withdraw_shares()` updates the user's share balance, and `internal_remove_withdrawal()` removes the user's submitted withdrawal structure. However, in the function `resolve_complete_queued_withdrawals()`, while the user's withdrawal is reinserted, the user's share balance is not updated accordingly, which is incorrect.

```
539 pub fn resolve_complete_queued_withdrawals(
```

```
540     &mut self,
541     withdrawals: Vec<Withdrawal>,
542     #[callback_result] success: Result<(), near_sdk::PromiseError>,
543 ) -> bool {
544     let success = success.is_ok();
545     if success {
546         for withdrawal in withdrawals.iter() {
547             Event::CompleteQueuedWithdrawal { withdrawal }.emit();
548         }
549         true
550     } else {
551         for withdrawal in withdrawals.iter() {
552             self.delegation_manager
553                 .internal_add_withdrawal(&withdrawal.staker, withdrawal);
554         }
555         false
556     }
557 }
```

Listing 2.1: delegation.rs

```
460 pub fn complete_queued_withdrawals(
461     &mut self,
462     withdrawals: Vec<Withdrawal>,
463 ) -> PromiseOrValue<bool> {
464     self.assert_contract_running();
465     assert_one_yocto();
466
467
468     require!(!withdrawals.is_empty(), ERR_EMPTY_WITHDRAWALS);
469     // get withdrawals from delegation manager
470     let withdrawals: Vec<Withdrawal> = withdrawals
471         .iter()
472         .map(|withdrawal| {
473             self.delegation_manager
474                 .internal_get_withdrawal(&withdrawal.staker, withdrawal.nonce)
475         })
476         .collect();
477
478
479     let withdrawer = env::predecessor_account_id();
480     // withdrawal strategies and amounts
481     let mut withdrawals_amounts: Vec<(StrategyId, Balance)> = Vec::new();
482     for withdrawal in withdrawals.iter() {
483         let mut withdrawal_amounts = self.delegation_manager.complete_queued_withdrawals(
484             &mut self.strategy_manager,
485             &withdrawal.staker,
486             &withdrawer,
487             withdrawal.nonce,
488         );
489         withdrawals_amounts.append(&mut withdrawal_amounts)
490     }
491 }
```



```
492
493     self.assert_storage_covered(&env::predecessor_account_id());
494
495
496     // Underlying tokens should be the same for all withdrawals
497     let token_id = self
498         .strategy_manager
499         .get_strategy(withdrawals_amounts[0].0)
500         .underlying_token;
501     let mut promise = Promise::new(token_id.clone());
502     for (strategy_id, amount) in withdrawals_amounts.iter() {
503         let underlying_token = self
504             .strategy_manager
505             .get_strategy(*strategy_id)
506             .underlying_token;
507         require!(
508             underlying_token == token_id,
509             ERR_STRATEGY_UNDERLYING_TOKEN_MISMATCH
510         );
511
512
513         // transfer tokens in batch action
514         promise = promise.function_call_weight(
515             "ft_transfer".to_string(),
516             serde_json::to_vec(&FtTransferArgs {
517                 receiver_id: withdrawer.clone(),
518                 amount: *amount,
519                 memo: Some(format!("withdraw from strategy #{}", strategy_id)),
520             })
521             .unwrap(),
522             ONE_YOCTO,
523             GAS_FOR_FT_TRANSFER,
524             GasWeight(1),
525         );
526     }
527     promise
528         .then(
529             Self::ext(env::current_account_id())
530                 .with_static_gas(
531                     GAS_FOR_RESOLVE_COMPLETE_QUEUED_WITHDRAWALS
532                     + GAS_FOR_ADD_ONE_WITHDRAWAL * withdrawals.len() as u64,
533                 )
534                 .resolve_complete_queued_withdrawals(withdrawals),
535         )
536         .into()
537 }
```

Listing 2.2: delegation.rs

```
257 pub fn complete_queued_withdrawals(
258     &mut self,
259     strategy_manager: &mut StrategyManager,
260     staker: &AccountId,
```

```
261     withdrawer: &AccountId,
262     nonce: Nonce,
263 ) -> Vec<(StrategyId, Balance)> {
264     let withdrawal = self.internal_get_withdrawal(staker, nonce);
265     require!(
266         withdrawal.start_at + self.minimum_withdrawal_delay_ms <= current_timestamp_ms(),
267         ERR_MINIMUM_WITHDRAWAL_DELAY_NOT_PASSED
268     );
269     require!(
270         withdrawer.clone() == withdrawal.withdrawer,
271         ERR_ONLY_WITHDRAWER
272     );
273
274
275     let mut withdrawal_amounts: Vec<(StrategyId, Balance)> = Vec::new();
276     for i in 0..withdrawal.strategies.len() {
277         let strategy_id = withdrawal.strategies[i];
278         require!(
279             withdrawal.start_at + self.internal_get_strategy_withdraw_delay_ms(&strategy_id)
280                 <= current_timestamp_ms(),
281             ERR_STRATEGY_WITHDRAWAL_DELAY_NOT_PASSED
282         );
283         let amount =
284             strategy_manager.withdraw_shares(staker, strategy_id, withdrawal.shares[i]);
285         withdrawal_amounts.push((strategy_id, amount));
286     }
287
288
289     self.internal_remove_withdrawal(staker, &nonce);
290
291
292     withdrawal_amounts
293 }
```

Listing 2.3: delegation.rs

```
250 pub fn withdraw_shares(
251     &mut self,
252     staker: &AccountId,
253     strategy_id: StrategyId,
254     shares: Shares,
255 ) -> Balance {
256     let mut strategy = self.get_strategy(strategy_id);
257     let balance = strategy.withdraw(staker, shares);
258     self.strategies.replace(strategy_id, &strategy.into());
259     balance
260 }
```

Listing 2.4: strategy.rs

```
396 fn internal_remove_withdrawal(&mut self, staker: &AccountId, nonce: &Nonce) {
397     let mut withdrawals = self.get_withdrawals(staker);
398     withdrawals.remove(nonce);
```

```
399     self.withdrawals.insert(staker, &withdrawals);
400 }
```

Listing 2.5: delegation.rs

Impact Lack of update to the user's share balance upon withdrawal failure prevents the user from retrying the withdrawal.

Suggestion Revise the logic to ensure that all states are correctly reverted when a user's withdrawal fails.

2.1.2 Lack of storage fee check

Severity Medium

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description Users invoke the function `complete_queued_withdrawals()` to remove the respective `withdrawal` request and withdraw the deposited assets. When the cross-contract invocation fails, the callback function `resolve_complete_queued_withdrawals()` will reinsert the user's `withdrawal` structure, which requires the additional storage fees. However, there is no storage fee check in the callback function.

```
442 pub fn queue_withdrawals(&mut self, queued_withdrawal_params: Vec<QueuedWithdrawalParams>) {
443     self.assert_contract_running();
444     assert_one_yocto();
445
446
447     require!(!queued_withdrawal_params.is_empty(), ERR_EMPTY_WITHDRAWALS);
448
449
450     let staker = env::predecessor_account_id();
451     self.delegation_manager.queue_withdrawals(
452         &mut self.strategy_manager,
453         &staker,
454         &queued_withdrawal_params,
455     );
456
457
458     self.assert_storage_covered(&env::predecessor_account_id());
459 }
```

Listing 2.6: delegation.rs

```
389 fn internal_add_withdrawal(&mut self, staker: &AccountId, withdrawal: &Withdrawal) {
390     let mut withdrawals = self.get_withdrawals(staker);
391     withdrawals.insert(&withdrawal.nonce, &withdrawal.into());
392     self.withdrawals.insert(staker, &withdrawals);
393 }
```

Listing 2.7: delegation.rs

```
460 pub fn complete_queued_withdrawals(
461     &mut self,
462     withdrawals: Vec<Withdrawal>,
463 ) -> PromiseOrValue<bool> {
464     self.assert_contract_running();
465     assert_one_yocto();
466
467
468     require!(!withdrawals.is_empty(), ERR_EMPTY_WITHDRAWALS);
469     // get withdrawals from delegation manager
470     let withdrawals: Vec<Withdrawal> = withdrawals
471         .iter()
472         .map(|withdrawal| {
473             self.delegation_manager
474                 .internal_get_withdrawal(&withdrawal.staker, withdrawal.nonce)
475         })
476         .collect();
477
478
479     let withdrawer = env::predecessor_account_id();
480     // withdrawal strategies and amounts
481     let mut withdrawals_amounts: Vec<(StrategyId, Balance)> = Vec::new();
482     for withdrawal in withdrawals.iter() {
483         let mut withdrawal_amounts = self.delegation_manager.complete_queued_withdrawals(
484             &mut self.strategy_manager,
485             &withdrawal.staker,
486             &withdrawer,
487             withdrawal.nonce,
488         );
489         withdrawals_amounts.append(&mut withdrawal_amounts)
490     }
491
492
493     self.assert_storage_covered(&env::predecessor_account_id());
494
495
496     // Underlying tokens should be the same for all withdrawals
497     let token_id = self
498         .strategy_manager
499         .get_strategy(withdrawals_amounts[0].0)
500         .underlying_token;
501     let mut promise = Promise::new(token_id.clone());
502     for (strategy_id, amount) in withdrawals_amounts.iter() {
503         let underlying_token = self
504             .strategy_manager
505             .get_strategy(*strategy_id)
506             .underlying_token;
507         require!(
508             underlying_token == token_id,
509             ERR_STRATEGY_UNDERLYING_TOKEN_MISMATCH
510         );
511     }
```

```

512
513     // transfer tokens in batch action
514     promise = promise.function_call_weight(
515         "ft_transfer".to_string(),
516         serde_json::to_vec(&FtTransferArgs {
517             receiver_id: withdrawer.clone(),
518             amount: *amount,
519             memo: Some(format!("withdraw from strategy #{}", strategy_id)),
520         })
521         .unwrap(),
522         ONE_YOCTO,
523         GAS_FOR_FT_TRANSFER,
524         GasWeight(1),
525     );
526 }
527 promise
528     .then(
529         Self::ext(env::current_account_id())
530             .with_static_gas(
531                 GAS_FOR_RESOLVE_COMPLETE_QUEUED_WITHDRAWALS
532                 + GAS_FOR_ADD_ONE_WITHDRAWAL * withdrawals.len() as u64,
533             )
534             .resolve_complete_queued_withdrawals(withdrawals),
535     )
536     .into()
537 }

```

Listing 2.8: delegation.rs

```

539 pub fn resolve_complete_queued_withdrawals(
540     &mut self,
541     withdrawals: Vec<Withdrawal>,
542     #[callback_result] success: Result<(), near_sdk::PromiseError>,
543 ) -> bool {
544     let success = success.is_ok();
545     if success {
546         for withdrawal in withdrawals.iter() {
547             Event::CompleteQueuedWithdrawal { withdrawal }.emit();
548         }
549         true
550     } else {
551         for withdrawal in withdrawals.iter() {
552             self.delegation_manager
553                 .internal_add_withdrawal(&withdrawal.staker, withdrawal);
554         }
555         false
556     }
557 }

```

Listing 2.9: delegation.rs

Impact User's storage fee may be insufficient while adding the withdrawal.

Suggestion Implement the storage check logic in the function

```
resolve_complete_queued_withdrawals().
```

2.1.3 Potential insufficient storage fee

Severity Medium

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description In the function `internal_storage_used_bytes()`, the `WITHDRAWAL_STORAGE_BYTES` is used to estimate the storage cost per Withdrawal for users. However, the `Withdrawal` struct includes dynamic array `strategies` and `shares`, and malicious users can inflate the storage cost of a `Withdrawal` significantly by padding it with a large number of elements in `strategies`. Storage costs for `Withdrawal` are much higher than `WITHDRAWAL_STORAGE_BYTES`.

```
140 fn internal_storage_used_bytes(&self, account_id: &AccountId) -> Balance {
141     MIN_STORAGE_BYTES
142     + self.delegation_manager.get_withdrawals(account_id).len() as u128
143     * WITHDRAWAL_STORAGE_BYTES
144 }
```

Listing 2.10: stroage.rs

```
70 pub struct Withdrawal {
71     /// The staker who starts the withdrawal
72     staker: AccountId,
73     /// The operator that the staker has delegated to
74     operator: Option<AccountId>,
75     /// The account that can complete the withdrawal and receive funds
76     withdrawer: AccountId,
77     // Nonce used to differentiate identical withdrawals
78     nonce: Nonce,
79     /// The withdrawal start time in milliseconds
80     start_at: TimestampMs,
81     /// The array of strategies that the withdrawal contains
82     #[serde(with = "u64_vec_format")]
83     strategies: Vec<StrategyId>,
84     /// The array of shares to withdraw for each strategy
85     #[serde(with = "u128_vec_format")]
86     shares: Vec<Shares>,
87 }
```

Listing 2.11: delegation.rs

Impact The storage cost of the user's `withdrawal` exceeds the `WITHDRAWAL_STORAGE_BYTES`.

Suggestion Check for duplicate elements in the dynamic array `strategies` within the `withdrawal`.

2.1.4 Lack of FT transfer fee charge

Severity Low

Status Confirmed

Introduced by [Version 1](#)

Description The function `complete_queued_withdrawals()` performs multiple FT transfers within a `for` loop, and each FT transfer requires attaching `ONE_YOCTO NEAR`. However, the contract only charges the fee from the user for once.

```
459  #[payable]
460  pub fn complete_queued_withdrawals(
461      &mut self,
462      withdrawals: Vec<Withdrawal>,
463  ) -> PromiseOrValue<bool> {
464      self.assert_contract_running();
465      assert_one_yocto();
466
467
468      require(!withdrawals.is_empty(), ERR_EMPTY_WITHDRAWALS);
469      // get withdrawals from delegation manager
470      let withdrawals: Vec<Withdrawal> = withdrawals
471          .iter()
472          .map(|withdrawal| {
473              self.delegation_manager
474                  .internal_get_withdrawal(&withdrawal.staker, withdrawal.nonce)
475          })
476          .collect();
477
478
479      let withdrawer = env::predecessor_account_id();
480      // withdrawal strategies and amounts
481      let mut withdrawals_amounts: Vec<(StrategyId, Balance)> = Vec::new();
482      for withdrawal in withdrawals.iter() {
483          let mut withdrawal_amounts = self.delegation_manager.complete_queued_withdrawals(
484              &mut self.strategy_manager,
485              &withdrawal.staker,
486              &withdrawer,
487              withdrawal.nonce,
488          );
489          withdrawals_amounts.append(&mut withdrawal_amounts)
490      }
491
492
493      self.assert_storage_covered(&env::predecessor_account_id());
494
495
496      // Underlying tokens should be the same for all withdrawals
497      let token_id = self
498          .strategy_manager
499          .get_strategy(withdrawals_amounts[0].0)
500          .underlying_token;
501      let mut promise = Promise::new(token_id.clone());
502      for (strategy_id, amount) in withdrawals_amounts.iter() {
503          let underlying_token = self
504              .strategy_manager
```

```

505         .get_strategy(*strategy_id)
506         .underlying_token;
507     require!(
508         underlying_token == token_id,
509         ERR_STRATEGY_UNDERLYING_TOKEN_MISMATCH
510     );
511
512
513     // transfer tokens in batch action
514     promise = promise.function_call_weight(
515         "ft_transfer".to_string(),
516         serde_json::to_vec(&FtTransferArgs {
517             receiver_id: withdrawer.clone(),
518             amount: *amount,
519             memo: Some(format!("withdraw from strategy #{}", strategy_id)),
520         })
521         .unwrap(),
522         ONE_YOCTO,
523         GAS_FOR_FT_TRANSFER,
524         GasWeight(1),
525     );
526 }
527 promise
528     .then(
529         Self::ext(env::current_account_id())
530             .with_static_gas(
531                 GAS_FOR_RESOLVE_COMPLETE_QUEUED_WITHDRAWALS
532                 + GAS_FOR_ADD_ONE_WITHDRAWAL * withdrawals.len() as u64,
533             )
534             .resolve_complete_queued_withdrawals(withdrawals),
535     )
536     .into()
537 }
```

Listing 2.12: delegation.rs

Impact The contract covers most of the [FT](#) transfer fees.

Suggestion Add [FT](#) transfer fee charge for each loop.

Feedback from the project Team stated that users have paid the minimum storage fee and the [FT](#) transfer cost in [Yocto NEAR](#) is negligible compared to the gas fee.

2.1.5 Lack of check in privileged functions

Severity Low

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description In the file [admin.rs](#), there are multiple privileged methods such as [change_owner_id\(\)](#), [pause_contract\(\)](#), and [resume_contract\(\)](#). These functions only verify that the caller is the contract owner but do not check that the attached [NEAR](#) equals 1 [yocto NEAR](#), which doesn't follow the best practice for implementing sensitive functions.


```
11 pub fn change_owner_id(&mut self, new_owner_id: AccountId) {
12     self.assert_owner();
13     self.owner_id = new_owner_id;
14 }
```

Listing 2.13: admin.rs

```
24 pub fn pause_contract(&mut self) {
25     self.assert_owner();
26     require!(!self.paused, ERR_ALREADY_PAUSED);
27     self.paused = true;
28 }
```

Listing 2.14: admin.rs

```
30 pub fn resume_contract(&mut self) {
31     self.assert_owner();
32     require!(self.paused, ERR_NOT_PAUSED);
33     self.paused = false;
34 }
```

Listing 2.15: admin.rs

Impact These privileged functions lack effective secondary confirmation and could be accidentally triggered.

Suggestion Add checks in above functions ensure the attached `NEAR` equals 1 `yocto NEAR`, and annotate these functions with `#[payable]`.

2.2 Additional Recommendation

2.2.1 Lack of check in function `storage_deposit()`

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description The function `storage_deposit()` lacks a check to verify if the contract is paused.

```
38 #[payable]
39 fn storage_deposit(
40     &mut self,
41     account_id: Option<AccountId>,
42     registration_only: Option<bool>,
43 ) -> StorageBalance {
44     let amount: Balance = env::attached_deposit();
45     let account_id = account_id.unwrap_or_else(env::predecessor_account_id);
46     let storage = self.internal_get_storage(&account_id);
47     let registration_only = registration_only.unwrap_or(false);
48     if let Some(mut storage) = storage {
49         if registration_only && amount > 0 {
50             Promise::new(env::predecessor_account_id()).transfer(amount);
51         } else {
```

```
52         storage.storage_balance += amount;
53         self.internal_set_storage(&account_id, storage);
54     }
55 } else {
56     let min_balance = self.storage_balance_bounds().min.0;
57     if amount < min_balance {
58         env::panic_str("The attached deposit is less than the minimum storage balance");
59     }
60
61
62     let mut storage = Storage::default();
63     if registration_only {
64         let refund = amount - min_balance;
65         if refund > 0 {
66             Promise::new(env::predecessor_account_id()).transfer(refund);
67         }
68         storage.storage_balance = min_balance;
69     } else {
70         storage.storage_balance = amount;
71     }
72     self.internal_set_storage(&account_id, storage);
73 }
74 self.internal_storage_balance_of(&account_id).unwrap()
75 }
```

Listing 2.16: stroage.rs

Suggestion Add a pause check in the function `storage_deposit()` .

2.2.2 Redundant code

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description In the file `storage.rs`, the line 151 in the function `assert_storage_covered()` is redundant, as the variable `_storage_balance_needed` is not used within the function. The same issue also occurs in the function `internal_storage_balance_of()`. The function `internal_storage_needed()` returns the storage fee cost for `MIN_STORAGE_BYTES` and the storage bytes needed to store the user's withdrawal structure. The function `storage_balance_bounds()` returns the storage fee cost for `MIN_STORAGE_BYTES`. Therefore, the return value of `internal_storage_needed()` is always greater than that of `storage_balance_bounds()`. Thus, using `max()` is redundant.

```
150 pub(crate) fn assert_storage_covered(&self, account_id: &AccountId) {
151     let _storage_balance_needed = self.internal_storage_needed(account_id);
152     assert!(
153         self.internal_storage_needed(account_id)
154         <= self.internal_unwrap_storage(account_id).storage_balance,
155         "Not enough storage balance"
156     );
157 }
```

Listing 2.17: storage.rs

```
129 fn internal_storage_balance_of(&self, account_id: &AccountId) -> Option<StorageBalance> {
130     self.internal_get_storage(account_id)
131         .map(|storage| StorageBalance {
132             total: storage.storage_balance.into(),
133             available: U128(storage.storage_balance.saturating_sub(std::cmp::max(
134                 self.internal_storage_needed(account_id),
135                 self.storage_balance_bounds().min.0,
136             ))),
137         })
138 }
```

Listing 2.18: storage.rs

```
140 fn internal_storage_needed(&self, account_id: &AccountId) -> Balance {
141     self.internal_storage_used_bytes(account_id) * env::storage_byte_cost()
142 }
```

Listing 2.19: storage.rs

```
146 fn internal_storage_used_bytes(&self, account_id: &AccountId) -> Balance {
147     MIN_STORAGE_BYTES
148     + self.delegation_manager.get_withdrawals(account_id).len() as u128
149     * WITHDRAWAL_STORAGE_BYTES
150 }
```

Listing 2.20: storage.rs

```
103 fn storage_balance_bounds(&self) -> StorageBalanceBounds {
104     StorageBalanceBounds {
105         min: U128(MIN_STORAGE_BYTES * env::storage_byte_cost()),
106         max: None,
107     }
108 }
```

Listing 2.21: storage.rs

Suggestion Remove the redundant code.

2.2.3 Lack of check in function `add_strategies_to_deposit_whitelist()` and `remove_strategies_from_deposit_whitelist()`

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description In the file `strategy.rs`, the contract owner can add `strategies` to or remove `strategies` from the deposit whitelist using the functions `add_strategies_to_deposit_whitelist()` and `remove_strategies_from_deposit_whitelist()`. However, these functions do not check whether the passed `strategies` are already on the

whitelist. Specifically, when invoking `add_strategies_to_deposit_whitelist()`, it should be ensured that the `strategies` are not already on the whitelist, and when invoking `remove_strategies_from_deposit_whitelist()`, it should be ensured that the `strategies` are on the whitelist.

```
328 pub fn add_strategies_to_deposit_whitelist(&mut self, strategies: Vec<U64>) {
329     self.assert_contract_running();
330     self.assert_owner();
331     assert_one_yocto();
332
333
334     self.strategy_manager
335         .add_strategies_to_deposit_whitelist(&strategies.iter().map(|s| s.0).collect());
336 }
```

Listing 2.22: strategy.rs

```
339 pub fn remove_strategies_from_deposit_whitelist(&mut self, strategies: Vec<U64>) {
340     self.assert_contract_running();
341     self.assert_owner();
342     assert_one_yocto();
343
344
345     self.strategy_manager
346         .remove_strategies_from_deposit_whitelist(&strategies.iter().map(|s| s.0).collect());
347 }
```

Listing 2.23: strategy.rs

Suggestion Add checks to ensure that the states of the `strategies` passed are as expected.

2.2.4 Lack of check in function `set_strategies_withdrawal_delay()`

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description In the file `delegation.rs`, the contract owner can invoke the function `set_strategies_withdrawal_delay()` to set the withdrawal delay for each strategy. However, the function does not check whether the input `StrategyId` is duplicated or whether the input `StrategyId` exists. Additionally, it only checks that the input `delays_ms` does not exceed `MAX_WITHDRAWAL_DELAY_MS`, without ensuring that `delays_ms` must be greater than or equal to `minimum_withdrawal_delay_ms`.

```
204 pub fn set_strategies_withdrawal_delay(
205     &mut self,
206     strategies: Vec<U64>,
207     delays_ms: Vec<TimestampMs>,
208 ) {
209     self.assert_contract_running();
210     self.assert_owner();
211     assert_one_yocto();
212 }
```

```
213
214     let strategy_ids = strategies.iter().map(|s| s.0).collect();
215     self.delegation_manager
216         .set_strategies_withdrawal_delay(&strategy_ids, &delays_ms);
217 }
```

Listing 2.24: delegation.rs

```
426 pub fn set_strategies_withdrawal_delay(
427     &mut self,
428     strategy_ids: &Vec<StrategyId>,
429     delays_ms: &Vec<TimestampMs>,
430 ) {
431     require!(
432         strategy_ids.len() == delays_ms.len(),
433         ERR_INPUT_LENGTH_MISMATCH
434     );
435     for i in 0..strategy_ids.len() {
436         require!(
437             delays_ms[i] <= MAX_WITHDRAWAL_DELAY_MS,
438             ERR_INVALID_WITHDRAW_DELAY_PERIOD
439         );
440         let strategy_id = strategy_ids[i];
441         let prev_withdrawal_delay_ms =
442             self.internal_get_strategy_withdraw_delay_ms(&strategy_id);
443         self.strategy_withdrawal_delay_ms
444             .insert(&strategy_id, &delays_ms[i]);
445         Event::SetStrategyWithdrawalDelay {
446             strategy_id: &U64(strategy_id),
447             prev_withdrawal_delay_ms: &prev_withdrawal_delay_ms,
448             new_withdrawal_delay_ms: &delays_ms[i],
449         }
450         .emit();
451     }
452 }
```

Listing 2.25: delegation.rs

Suggestion Add checks to ensure that the input parameters match expectations.

2.3 Note

2.3.1 Potential centralization risk

Introduced by [Version 1](#)

Description In the protocol, various checks exist, and the contract owner can modify these limit checks. If the owner's private key is lost or maliciously exploited, it could lead to losses for the protocol.

Feedback from the project Team guarantees that the contract owner is a [DAO](#) and requires confirmation from both the committee and the community before making any configuration changes.

