BlockSec

Security Audit Report for Paras NFT Contract

Date: September 23, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.10.05 14:20:49 +08'00'

Contents

Report Manifest

Item	Description
Client	Paras
Target	Paras NFT Contract

Version History

Version	Date	Description
1.0	September 23, 2022	First Release

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 the **Paras NFT** contract [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 (`Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

s Project		Commit SHA
Paras NFT Contract	`Version 1`	`8974748d4deeaed8c1a2351ab63e3950907b0485`
	`Version 2`	`4627338269f8b13db4e56244d0d873f4654a978b`

Note that, we did **NOT** audit all the modules in the repository. The modules covered by this audit report include **paras-nft-contract/src** folder contract only. Specifically, the file covered in this audit include:

- event.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.

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] https://github.com/ParasHQ/paras-nft-contract

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
* 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		**Likelihood**	
		High	*Low*
	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 **four** potential issues. We have **six** recommendations and **one** note.

- High Risk: 0
- Medium Risk: 0
- Low Risk: 4
- Recommendations: 6
- Notes: 1

ID	Severity	Description	Category	Status
1	Low	Non-Mintable NFT with a Selling Price	Software Security	Fixed
2	Low	Potential Inconsistent Transaction Fee	DeFi Security	Confirmed
3	Low	Incomplete NFT Token Burning Mechanism	NFT Security	Confirmed
4	Low	Transaction Fee Bypass with Direct NFT Minting	NFT Security	Confirmed
5	-	Potential Centralization Problem	Recommendation	Confirmed
6	-	Improper NFT Series MetaData Query	Recommendation	Fixed
7	-	Redundant Code (I)	Recommendation	Acknowledged
8	-	Redundant Code (II)	Recommendation	Confirmed
9	-	Redundant Function Parameter	Recommendation	Fixed
10	-	Storage Optimization	Recommendation	Fixed
11	-	Assumption on the Secure Implementation of Dependencies	Notes	Confirmed

The details are provided in the following sections.

2.1 Software Security

2.1.1 Non-Mintable NFT with a Selling Price

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `nft_decrease_series_copies()`, the price of the NFT series is not set as `None` when it is not mintable anymore (lines 615-617) due the deduction of the series copies, which is inconsistent with the implementation of another function `_nft_mint_series()` (lines 539-542).

```
592 #[payable]
593 pub fn nft_decrease_series_copies(
594     &mut self,
595     token_series_id: TokenSeriesId,
596     decrease_copies: U64
597 ) -> U64 {
598     assert_one_yocto();
599
600     let mut token_series = self.token_series_by_id.get(&token_series_id).expect("Token series
            not exist");
601     assert_eq!(
        env::predecessor_account_id(),
        token_series.creator_id,
        "Paras: Creator only"
    );

    let minted_copies = token_series.tokens.len();
    let copies = token_series.metadata.copies.unwrap();

    assert!(
        (copies - decrease_copies.0) >= minted_copies,
        "Paras: cannot decrease supply, already minted : {}", minted_copies
    );

    let is_non_mintable = if (copies - decrease_copies.0) == minted_copies {
        token_series.is_mintable = false;
        true
    } else {
        false
    };

    token_series.metadata.copies = Some(copies - decrease_copies.0);

    self.token_series_by_id.insert(&token_series_id, &token_series);
    env::log(
        json!({
            "type": "nft_decrease_series_copies",
            "params": {
                "token_series_id": token_series_id,
                "copies": U64::from(token_series.metadata.copies.unwrap()),
                "is_non_mintable": is_non_mintable,
            }
        })
        .to_string()
        .as_bytes(),
    );
    U64::from(token_series.metadata.copies.unwrap())
}
```

Listing 2.1: paras-nft-contract/src/lib.rs

```
fn _nft_mint_series(
  &mut self,
  token_series_id: TokenSeriesId,
  receiver_id: AccountId
) -> TokenId {
  let mut token_series = self.token_series_by_id.get(&token_series_id).expect("Paras: Token
      series not exist");
  assert!(
      token_series.is_mintable,
      "Paras: Token series is not mintable"
  );

  let num_tokens = token_series.tokens.len();
  let max_copies = token_series.metadata.copies.unwrap_or(u64::MAX);
  assert!(num_tokens < max_copies, "Series supply maxed");

  if (num_tokens + 1) >= max_copies {
      token_series.is_mintable = false;
      token_series.price = None;
  }

  let token_id = format!("{}{}{}", &token_series_id, TOKEN_DELIMETER, num_tokens + 1);
  token_series.tokens.insert(&token_id);
  self.token_series_by_id.insert(&token_series_id, &token_series);
```

Listing 2.2: paras-nft-contract/src/lib.rs

Impact There will be some NFT series with certain available prices, but cannot be bought by the buyers.

Suggestion Remove the `TokenSeries`'s price if it is not mintable in function `nft_decrease_series_copies()`.

2.2 DeFi Security

2.2.1 Potential Inconsistent Transaction Fee

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The `market_data_transaction_fee` for a certain series will not be updated unless the function `nft_set_series_price()` is invoked by the creator (lines 670-672).

```
640 #[payable]
641 pub fn nft_set_series_price(&mut self, token_series_id: TokenSeriesId, price: Option<U128>) ->
        Option<U128> {
642     assert_one_yocto();
643
644     let mut token_series = self.token_series_by_id.get(&token_series_id).expect("Token series
            not exist");
645     assert_eq!(
646         env::predecessor_account_id(),
647         token_series.creator_id,
648         "Paras: Creator only"
649     );
650
651     assert_eq!(
652         token_series.is_mintable,
653         true,
654         "Paras: token series is not mintable"
655     );
656
657     if price.is_none() {
658         token_series.price = None;
659     } else {
660         assert!(
```

```rust
            price.unwrap().0 < MAX_PRICE,
            "Paras: price higher than {}",
            MAX_PRICE
        );
        token_series.price = Some(price.unwrap().0);
    }

    self.token_series_by_id.insert(&token_series_id, &token_series);

    // set market data transaction fee
    let current_transaction_fee = self.calculate_current_transaction_fee();
    self.market_data_transaction_fee.transaction_fee.insert(&token_series_id, &
        current_transaction_fee);

    env::log(
        json!({
            "type": "nft_set_series_price",
            "params": {
                "token_series_id": token_series_id,
                "price": price,
                "transaction_fee": current_transaction_fee.to_string()
            }
        })
        .to_string()
        .as_bytes(),
    );
    return price;
}
```

Listing 2.3: paras-nft-contract/src/lib.rs

Impact Buyers may have to pay the treasury with the outdated transaction fee even if the contract's current transaction fee `(Contract.transaction_fee)` is already changed by function `set_transaction_fee()`.

```rust
#[payable]
pub fn set_transaction_fee(&mut self, next_fee: u16, start_time: Option<TimestampSec>) {
    assert_one_yocto();
    assert_eq!(
        env::predecessor_account_id(),
        self.tokens.owner_id,
        "Paras: Owner only"
    );

    assert!(
        next_fee < 10_000,
        "Paras: transaction fee is more than 10_000"
    );

    if start_time.is_none() {
        self.transaction_fee.current_fee = next_fee;
        self.transaction_fee.next_fee = None;
        self.transaction_fee.start_time = None;
        return
    } else {
        let start_time: TimestampSec = start_time.unwrap();
        assert!(
            start_time > to_sec(env::block_timestamp()),
            "start_time is less than current block_timestamp"
        );
        self.transaction_fee.next_fee = Some(next_fee);
        self.transaction_fee.start_time = Some(start_time);
    }
}
```

Listing 2.4: paras-nft-contract/src/lib.rs

Suggestion Calculate the treasury fee based on the current default transaction fee `(Contract.transaction-_fee)` in function `nft_buy()`.

```
#[payable]
pub fn nft_buy(
    &mut self,
    token_series_id: TokenSeriesId
) -> TokenId {
    let initial_storage_usage = env::storage_usage();
    let attached_deposit = env::attached_deposit();
    let receiver_id = env::predecessor_account_id();
    let token_series = self.token_series_by_id.get(&token_series_id).expect("Paras: Token
        series not exist");
    let price: u128 = token_series.price.expect("Paras: not for sale");
    assert!(
        attached_deposit >= price,
        "Paras: attached deposit is less than price : {}",
        price
    );
    let token_id: TokenId = self._nft_mint_series(token_series_id.clone(), receiver_id.
        to_string());

    let for_treasury = price as u128 * self.calculate_market_data_transaction_fee(&
        token_series_id) / 10_000u128;
    let price_deducted = price - for_treasury;
    Promise::new(token_series.creator_id).transfer(price_deducted);

    if for_treasury != 0 {
        Promise::new(self.treasury_id.clone()).transfer(for_treasury);
    }

    refund_deposit(env::storage_usage() - initial_storage_usage, price);

    NearEvent::log_nft_mint(
        receiver_id.to_string(),
        vec![token_id.clone()],
        Some(json!({"price": price.to_string()}).to_string())
    );

    token_id
}
```

Listing 2.5: paras-nft-contract/src/lib.rs

Feedback from the Project This is by design. The transaction fee is determined when the price was set.

2.3 NFT Security

2.3.1 Incomplete NFT Token Burning Mechanism

Severity Low

Status Confirmed

Introduced by `Version 1`

Description When the function `nft_burn()` is invoked, the `token_id` of the burnt NFT token will not be removed from the *UnorderedSet* `TokenSeries.tokens`, which means that the supply of the corresponding NFT series will not be reduced.

```rust
#[payable]
pub fn nft_burn(&mut self, token_id: TokenId) {
    assert_one_yocto();

    let owner_id = self.tokens.owner_by_id.get(&token_id).unwrap();
    assert_eq!(
        owner_id,
        env::predecessor_account_id(),
        "Token owner only"
    );

    if let Some(next_approval_id_by_id) = &mut self.tokens.next_approval_id_by_id {
        next_approval_id_by_id.remove(&token_id);
    }

    if let Some(approvals_by_id) = &mut self.tokens.approvals_by_id {
        approvals_by_id.remove(&token_id);
    }

    if let Some(tokens_per_owner) = &mut self.tokens.tokens_per_owner {
        let mut token_ids = tokens_per_owner.get(&owner_id).unwrap();
        token_ids.remove(&token_id);
        tokens_per_owner.insert(&owner_id, &token_ids);
    }

    if let Some(token_metadata_by_id) = &mut self.tokens.token_metadata_by_id {
        token_metadata_by_id.remove(&token_id);
    }

    self.tokens.owner_by_id.remove(&token_id);

    NearEvent::log_nft_burn(
        owner_id,
        vec![token_id],
        None,
        None,
    );
}
```

Listing 2.6: paras-nft-contract/src/lib.rs

Impact The burnt NFT Token cannot be minted again. This is because the `token_id` of the newly minted token is based on the length of the *UnorderedSet* `TokenSeries.tokens` (line 535) and its length will only increase.

```
524    fn _nft_mint_series(
525        &mut self,
526        token_series_id: TokenSeriesId,
527        receiver_id: AccountId
528    ) -> TokenId {
529        let mut token_series = self.token_series_by_id.get(&token_series_id).expect("Paras: Token series not exist");
530        assert!(
531            token_series.is_mintable,
532            "Paras: Token series is not mintable"
533        );
534
535        let num_tokens = token_series.tokens.len();
536        let max_copies = token_series.metadata.copies.unwrap_or(u64::MAX);
537        assert!(num_tokens < max_copies, "Series supply maxed");
538
539        if (num_tokens + 1) >= max_copies {
540            token_series.is_mintable = false;
541            token_series.price = None;
542        }
543
544        let token_id = format!("{}{}{}", &token_series_id, TOKEN_DELIMETER, num_tokens + 1);
545        token_series.tokens.insert(&token_id);
546        self.token_series_by_id.insert(&token_series_id, &token_series);
```

Listing 2.7: paras-nft-contract/src/lib.rs

Suggestion Remove the burnt NFT's `token_id` from the *UnorderedSet* `TokenSeries.tokens` in function `nft_burn()` and implement a reasonable method to generate the `token_id` of the newly minted NFT in function `_nft_mint_series()`.

Feedback from the Project This is by design, because the supply also includes burnt tokens.

2.3.2 Transaction Fee Bypass with Direct NFT Minting

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In function `nft_mint()`, the NFT series creators can mint NFTs without paying the corresponding `market_data_transaction_fee`, which allows the trade to be made offline.

```rust
#[payable]
pub fn nft_mint(
    &mut self,
    token_series_id: TokenSeriesId,
    receiver_id: ValidAccountId
) -> TokenId {
    let initial_storage_usage = env::storage_usage();

    let token_series = self.token_series_by_id.get(&token_series_id).expect("Paras: Token
        series not exist");
    assert_eq!(env::predecessor_account_id(), token_series.creator_id, "Paras: not creator");
    let token_id: TokenId = self._nft_mint_series(token_series_id, receiver_id.to_string());

    refund_deposit(env::storage_usage() - initial_storage_usage, 0);

    NearEvent::log_nft_mint(
        receiver_id.to_string(),
        vec![token_id.clone()],
        None,
    );

    token_id
}
```

Listing 2.8: paras-nft-contract/src/lib.rs

Impact NFTs can be minted without paying the transaction fee.

Suggestion It is suggested to calculate and charge transaction fee in function `nft_mint()`.

Feedback from the Project This is by design. Since the mint here was done by the creator itself, it doesn't need to go through payment which does not require transaction fee.

2.4 Additional Recommendation

2.4.1 Potential Centralization Problem

Status Confirmed

Introduced by `Version 1`

Description The privileged account `Contract.tokens.owner_id` has the ability to configure some of the system parameters (e.g., `Contract.transaction_fee` and `Contract.treasury_id)`. Additionally, the person who has the full access key of this contract could transfer assets out (e.g., NEARs) and upgrade the contract directly.

Suggestion It's suggested to remove the full access key of the contract from the blockchain (via `DeleteKey` transaction) and implement the privileged upgrade function. Besides, a decentralization design is also recommended to be introduced in the contract. The privileged roles are suggested to be transferred to a multi-signature account or DAO.

Feedback from the Project Will move the ownership to multi-sig.

2.4.2 Improper NFT Series MetaData Query

Status Fixed in `Version 2`

Introduced by `Version 1`

Description According to the current contract implementation, the transaction fee is different for each NFT series (line 111). In this case, it is necessary to return a specific transaction fee in the view function `nft_get_series()` instead of `None` (line 776).

```
109#[derive(BorshDeserialize, BorshSerialize, PanicOnDefault)]
110pub struct MarketDataTransactionFee {
111    pub transaction_fee: UnorderedMap<TokenSeriesId, u128>
112}
```

Listing 2.9: paras-nft-contract/src/lib.rs

```
754    pub fn nft_get_series(
755        &self,
756        from_index: Option<U128>,
757        limit: Option<u64>,
758    ) -> Vec<TokenSeriesJson> {
759        let start_index: u128 = from_index.map(From::from).unwrap_or_default();
760        assert!(
761            (self.token_series_by_id.len() as u128) > start_index,
762            "Out of bounds, please use a smaller from_index."
763        );
764        let limit = limit.map(|v| v as usize).unwrap_or(usize::MAX);
765        assert_ne!(limit, 0, "Cannot provide limit of 0.");
766
767        self.token_series_by_id
768            .iter()
769            .skip(start_index as usize)
770            .take(limit)
771            .map(|(token_series_id, token_series)| TokenSeriesJson{
772                token_series_id,
773                metadata: token_series.metadata,
774                creator_id: token_series.creator_id,
775                royalty: token_series.royalty,
776                transaction_fee: None
777            })
778            .collect()
779    }
```

Listing 2.10: paras-nft-contract/src/lib.rs

Suggestion Return a specific transaction fee for each NFT series in the view function `nft_get_series()`.

2.4.3 Redundant Code (I)

Status Acknowledged

Introduced by `Version 1`

Description According to the current implementation of contract, the `market_data_transaction_fee` associated with a specific NFT series won't be `None` when the series is created. However, both function `calculate_market_data_transaction_fee()` and function `get_market_data_transaction_fee()` assume that the corresponding `market_data_transaction_fee` of the input `token_series_id` could be `None`, and implement the inconsistent logic to fallback the transaction fee to default, which is redundant.

```rust
pub fn calculate_market_data_transaction_fee(&mut self, token_series_id: &TokenSeriesId) -> u128{
    if let Some(transaction_fee) = self.market_data_transaction_fee.transaction_fee.get(&token_series_id){
        return transaction_fee;
    }

    // fallback to default transaction fee
    self.calculate_current_transaction_fee()
}
```

Listing 2.11: paras-nft-contract/src/lib.rs

```rust
pub fn get_market_data_transaction_fee (&self, token_series_id: &TokenId) -> u128{
    if let Some(transaction_fee) = self.market_data_transaction_fee.transaction_fee.get(&token_series_id){
        return transaction_fee;
    }
    // fallback to default transaction fee
    self.transaction_fee.current_fee as u128
}
```

Listing 2.12: paras-nft-contract/src/lib.rs

Suggestion There is no need to fallback the transaction fee to default in function `calculate_market_data_transaction_fee()` and function `get_market_data_transaction_fee()`.

Feedback from the Project This is by design. We implement this functionality after many NFT series have been created, which explains why the `market_data_transaction_fee` could be `None`.

2.4.4 Redundant Code (II)

Status Confirmed

Introduced by `Version 1`

Description In function `nft_buy()`, it is unnecessary to check the amount of the attached NEARs (lines 421-425). If the attached NEARs cannot pay the price of the NFT plus the required storage fee, the transaction will throw into a panic in function `refund_deposit()` (line 436).

```rust
411    #[payable]
412    pub fn nft_buy(
413        &mut self,
414        token_series_id: TokenSeriesId
415    ) -> TokenId {
416        let initial_storage_usage = env::storage_usage();
417        let attached_deposit = env::attached_deposit();
418        let receiver_id = env::predecessor_account_id();
419        let token_series = self.token_series_by_id.get(&token_series_id).expect("Paras: Token series not exist");
420        let price: u128 = token_series.price.expect("Paras: not for sale");
421        assert!(
422            attached_deposit >= price,
423            "Paras: attached deposit is less than price : {}",
424            price
425        );
426        let token_id: TokenId = self._nft_mint_series(token_series_id.clone(), receiver_id.to_string());
427
428        let for_treasury = price as u128 * self.calculate_market_data_transaction_fee(&token_series_id) / 10_000u128;
429        let price_deducted = price - for_treasury;
430        Promise::new(token_series.creator_id).transfer(price_deducted);
431
432        if for_treasury != 0 {
433            Promise::new(self.treasury_id.clone()).transfer(for_treasury);
434        }
435
436        refund_deposit(env::storage_usage() - initial_storage_usage, price);
437
438        NearEvent::log_nft_mint(
439            receiver_id.to_string(),
440            vec![token_id.clone()],
441            Some(json!({"price": price.to_string()}).to_string())
442        );
443
444        token_id
445    }
```

Listing 2.13: paras-nft-contract/src/lib.rs

```
1138fn refund_deposit(storage_used: u64, extra_spend: Balance) {
1139    let required_cost = env::storage_byte_cost() * Balance::from(storage_used);
1140    let attached_deposit = env::attached_deposit() - extra_spend;
1141
1142    assert!(
1143        required_cost <= attached_deposit,
1144        "Must attach {} yoctoNEAR to cover storage",
1145        required_cost,
1146    );
1147
1148    let refund = attached_deposit - required_cost;
1149    if refund > 1 {
1150        Promise::new(env::predecessor_account_id()).transfer(refund);
1151    }
1152}
```

Listing 2.14: paras-nft-contract/src/lib.rs

Suggestion Remove the redundant assertion in function `nft_buy()` (lines 421-425).

2.4.5 Redundant Function Parameter

Status Fixed in Version 2

Introduced by Version 1

Description It is unnecessary to pass the parameter creator_id to the function nft_create_series() as the creator_id of this NFT series will eventually be set to env::predecessor_account_id(), which does not depend on the input creator_id.

```rust
#[payable]
pub fn nft_create_series(
    &mut self,
    creator_id: Option<ValidAccountId>,
    token_metadata: TokenMetadata,
    price: Option<U128>,
    royalty: Option<HashMap<AccountId, u32>>,
) -> TokenSeriesJson {
    let initial_storage_usage = env::storage_usage();
    let caller_id = env::predecessor_account_id();

    if creator_id.is_some() {
        assert_eq!(creator_id.unwrap().to_string(), caller_id, "Paras: Caller is not creator_id
            ");
    }

    let token_series_id = format!("{}", (self.token_series_by_id.len() + 1));

    assert!(
        self.token_series_by_id.get(&token_series_id).is_none(),
        "Paras: duplicate token_series_id"
    );
```

Listing 2.15: paras-nft-contract/src/lib.rs

Suggestion Remove the redundant parameter creator_id of function nft_create_series() for code optimization.

2.4.6 Storage Optimization

Status Fixed in Version 2

Introduced by Version 1

Description In function nft_burn(), if the set token_ids is empty after the removal, it's recommended to remove the *key/value* of this user from the *LookupMap* Contract.tokens.tokens_per_owner to free up the storage space.

```rust
#[payable]
pub fn nft_burn(&mut self, token_id: TokenId) {
    assert_one_yocto();

    let owner_id = self.tokens.owner_by_id.get(&token_id).unwrap();
    assert_eq!(
        owner_id,
            env::predecessor_account_id(),
            "Token owner only"
        );

        if let Some(next_approval_id_by_id) = &mut self.tokens.next_approval_id_by_id {
            next_approval_id_by_id.remove(&token_id);
        }

        if let Some(approvals_by_id) = &mut self.tokens.approvals_by_id {
            approvals_by_id.remove(&token_id);
        }

        if let Some(tokens_per_owner) = &mut self.tokens.tokens_per_owner {
            let mut token_ids = tokens_per_owner.get(&owner_id).unwrap();
            token_ids.remove(&token_id);
            tokens_per_owner.insert(&owner_id, &token_ids);
        }

        if let Some(token_metadata_by_id) = &mut self.tokens.token_metadata_by_id {
            token_metadata_by_id.remove(&token_id);
        }

        self.tokens.owner_by_id.remove(&token_id);

        NearEvent::log_nft_burn(
            owner_id,
            vec![token_id],
            None,
            None,
        );
    }
```

Listing 2.16: paras-nft-contract/src/lib.rs

Suggestion Remove the empty set `token_ids` from the *LookupMap* `Contract.tokens.tokens_per_owner` in time.

2.5 Notes

2.5.1 Assumption on the Secure Implementation of Dependencies

Status Confirmed

Introduced by `Version 1`

Description This `PARAS_NFT_CONTRACT` is built based on the crates `near-sdk` (version 3.1.0) and `near-contract-standards` (version 3.2.0).

The required interfaces and the basic functionality listed below are provided in the contract:

* NEP-171 (Non-Fungible Token Core Functionality)
* NEP-178 (Non-Fungible Token Approval Management)
* NEP-181 (Non-Fungible Token Enumeration)

- ∗ NEP-177 (Non-Fungible Token Metadata Standard)
- ∗ NEP-199 (Non-Fungible Token Royalties and Payouts)

In this audit, we assume the standard library provided by NEAR-SDK-RS [1] (i.e., `near_contract_standards`) has no security issues.

[1] https://github.com/near/near-sdk-rs