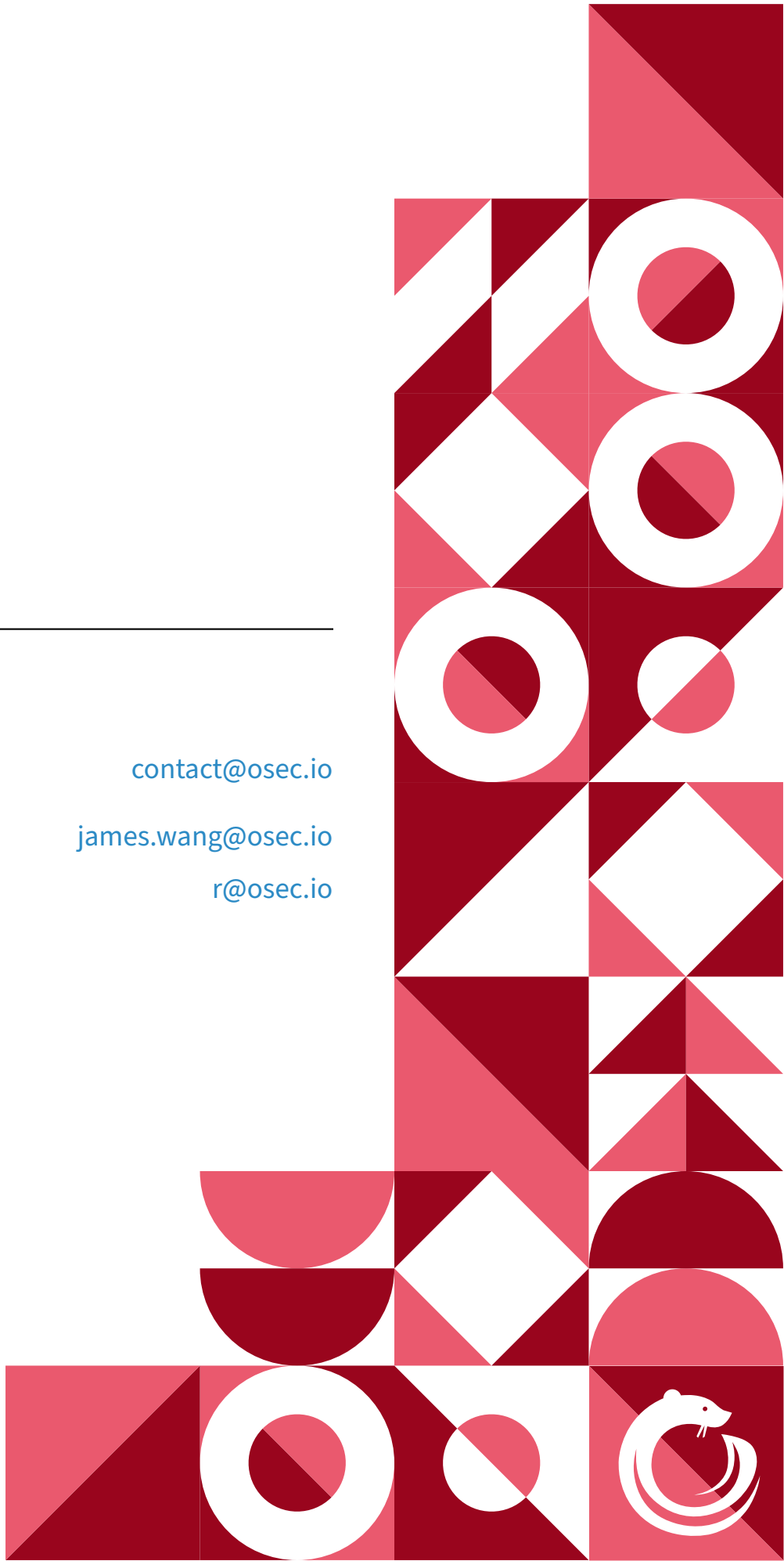


Mintbase

Audit

Presented by:

OtterSec contact@osec.io

James Wang james.wang@osec.io

Robert Chen r@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Mintbase engaged OtterSec to perform an assessment of the `mb-contract` programs. This assessment was conducted between March 20th and March 29th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 14 findings total.

In particular, we identified an issue with an incorrect NFT ownership management during transfers (OS-MTB-ADV-00, OS-MTB-ADV-01), as well as the possibility of native asset draining through storage fee refund (OS-MTB-ADV-02), and a potential user loss of NFTs or funds due to asynchronous receipt processing (OS-MTB-ADV-03, OS-MTB-ADV-04).

We also made recommendations around removing redundant code to reduce gas (OS-MTB-SUG-00), fixing documentation and implementation mismatches (OS-MTB-SUG-01), and best coding practices regarding storage fee management (OS-MTB-SUG-02).

02 | Scope

The source code was delivered to us in a git repository at github.com/Mintbase/mb-contracts. This audit was performed against commit aedb2c4.

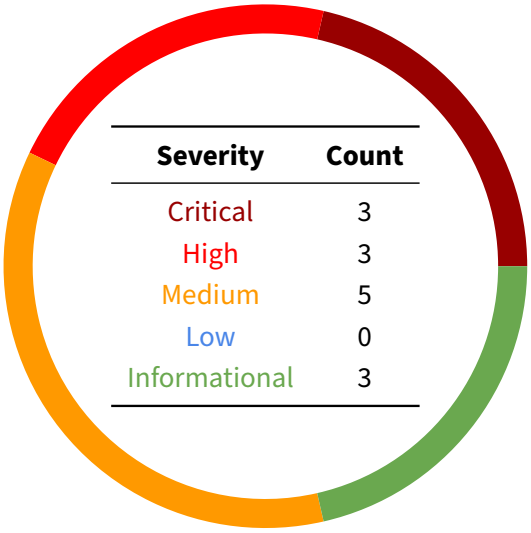
A brief description of the programs is as follows.

Name	Description
mb-factory	Factory contract responsible for new mb-store deployments.
mb-store	NFT implementation by Mintbase.
mb-legacy-market	Old implementation of NFT market that only allows trading between NFTs and native assets. Supports simple sales and rolling auctions.
mb-interop-market	New implementation of NFT market that supports both trading between NFTs and FTs and between NFTs and native assets while only supporting simple sales.

03 | Findings

Overall, we reported 14 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	3
High	3
Medium	5
Low	0
Informational	3

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-MTB-ADV-00	Critical	Resolved	`nft_transfer_call` does not assign the NFT to the receiver.
OS-MTB-ADV-01	Critical	Resolved	The `MintbaseStore` does not return NFTs to the original owner in `nft_transfer_call` upon `nft_on_transfer` failures.
OS-MTB-ADV-02	Critical	Resolved	`nft_revoke` incorrectly refunds storage fees to the caller.
OS-MTB-ADV-03	High	Resolved	`kick_token` does not respect token listing locks and may result in a loss of user and market funds.
OS-MTB-ADV-04	High	Resolved	`remove_offer` in the interop market may remove listing with offers, leading to a potential loss of user funds.
OS-MTB-ADV-05	High	Resolved	`nft_resolve_payout_ft` does not account for fees during `ft_transfer` and may result in a user loss of NFTs.
OS-MTB-ADV-06	Medium	Resolved	`nft_approve` may be abused to inflate gas costs of certain `MintbaseStore` functions and render those unavailable.
OS-MTB-ADV-07	Medium	Resolved	`MintbaseStore` will not be able to revoke minter privileges due to the unpayable `withdraw_minter` function.
OS-MTB-ADV-08	Medium	Resolved	NFT `approvals` and `split_owners` are not properly restored on transfer failures.

OS-MTB-ADV-09	Medium	Resolved	Storage fees for NFT listings in the marketplace are not collected from the caller, allowing the lock of legacy market funds.
OS-MTB-ADV-10	Medium	Resolved	`MintbaseStore` does not charge callers for potential storage fees, allowing attackers to lock all funds.

OS-MTB-ADV-00 [crit] | NFT Not Transferred In Transfer With Call

Description

`NEP-171` specifies that `nft_transfer_call` is required to transfer ownership of `token_id` NFTs from `previous_owner_id` to `receiver_id`. However, the `MintbaseStore` does not follow `NEP-171`'s specifications and delays the NFT transfer until the callback function `nft_resolve_transfer`.

This results in confusion within calls to `nft_on_transfer` and potentially leads to a loss of NFTs for the receiver.

Remediation

Transfer NFTs to `receiver` before locking the token.

mb-store/src/core.rs DIFF

```
/// Transfer-and-call function as specified by [NEP-171](https://nomicon.io/Standards/Tokens/NonFungibleToken/Core).
#[payable]
pub fn nft_transfer_call(
    &mut self,
    receiver_id: AccountId,
    token_id: U64,
    approval_id: Option<u64>,
    msg: String,
) -> Promise {
    assert_one_yocto();
    let token_idu64 = token_id.into();
    let mut token = self.nft_token_internal(token_idu64);
    let pred = env::predecessor_account_id();
    assert_token_unloaned!(token);
    assert_token_owned_or_approved!(token, &pred, approval_id);
    // prevent race condition, temporarily lock-replace owner
    let owner_id = AccountId::new_unchecked(token.owner_id.to_string());
+   self.transfer_internal(&mut token, receiver_id.clone(), true);
+   log_nft_transfer(
+       &receiver_id,
+       token.id,
+       &None,
+       owner_id.to_string(),
+   );
    self.lock_token(&mut token);

    ext_nft_on_transfer::ext(receiver_id.clone())
        .with_static_gas(gas::NFT_TRANSFER_CALL)
        .nft_on_transfer(pred, owner_id.clone(), token_id, msg)
        .then(
            store_self::ext(env::current_account_id())
                .with_static_gas(gas::NFT_TRANSFER_CALL)
                .nft_resolve_transfer(
                    owner_id,
                    receiver_id,
                    token_id.0.to_string(),
                    None,
                ),
        )
}
```

Patch

Resolved in 3563e5e and 89f1661.

OS-MTB-ADV-01 [crit] | NFT Not Returned On Transfer Failure

Description

`nft_resolve_transfer` is a callback function responsible for handling state rollback upon failure in `nft_on_transfer` or `nft_transfer_call`. A crucial aspect in rolling back states is to return NFTs to its `original_owner`.

However, `nft_resolve_transfer` calls `transfer_internal` with an incorrect recipient and fails to return the NFT back to the original owner properly.

mb-store/src/core.rs RUST

```rust
#[private]
pub fn nft_resolve_transfer(
    ...
) -> bool {
    ...
    if !must_revert {
        true
    } else {
        self.transfer_internal(&mut token, receiver_id.clone(), true);
        log_nft_transfer(
            &receiver_id,
            token_id_u64,
            &None,
            owner_id.to_string(),
        );
        false
    }
}
```

Remediation

Transfer NFTs to `owner_id` instead of `receiver_id`.

mb-store/src/core.rs DIFF

```
if !must_revert {
    true
} else {
-   self.transfer_internal(&mut token, receiver_id.clone(), true);
+   self.transfer_internal(&mut token, owner_id.clone(), true);
    log_nft_transfer(
-       &receiver_id,
+       &owner_id,
        token_id_u64,
        &None,
-       owner_id.to_string(),
+       receiver_id.to_string(),
    );
    false
}
```

Patch

Resolved in 6a69314 and 23ecd80.

OS-MTB-ADV-02 [crit] | MintbaseStore Funds Drainage Through Revoke

Description

Upon approval revokes, `AccountIds` are removed from `token.approval` and reduce storage usage. Then, `MintbaseStore` attempts to refund the user with the storage fees previously deposited when calling `nft_approve`.

However, since the highlighted `transfer` is called even when a revoked approval does not exist, it is possible for attackers to repeatedly call the function and steal `self.storage_costs.common` one transaction at a time until the `MintbaseStore` is completely drained of funds.

mb-store/src/approval.rs RUST

```rust
pub fn nft_revoke(
    &mut self,
    token_id: U64,
    account_id: AccountId,
) -> Promise {
    ...
    if token.approvals.remove(&account_id).is_some() {
        ...
    }

    Promise::new(env::predecessor_account_id())
        .transfer(self.storage_costs.common)
}
```

Remediation

Move `transfer` inside the `if` block, and only refund if approval to revoke exists.

mb-store/src/approval.rs DIFF

```
  pub fn nft_revoke(
      &mut self,
      token_id: U64,
      account_id: AccountId,
- ) -> Promise {
+ ) -> PromiseOrValue(()) {
      ...
      if token.approvals.remove(&account_id).is_some() {
          self.tokens.insert(&token_idu64, &token);
          log_revoke(token_idu64, &account_id);
+         PromiseOrValue::Promise(
+             Promise::new(env::predecessor_account_id())
+                 .transfer(self.storage_costs.common)
+         )
+     } else {
+         PromiseOrValue::Value(())
      }
-
-     Promise::new(env::predecessor_account_id())
-         .transfer(self.storage_costs.common)
  }
```

Patch

Resolved in cdedb98.

OS-MTB-ADV-03 [high] | Kick Token Does Not Respect Listing Lock

Description

The legacy marketplace locks the token when an offer is accepted, and no actions should be able to tamper with the token state before the offer is properly processed.

mb-legacy-market/src/offer.rs RUST

```
fn help_transfer(&mut self, token_key: &TokenKey, mut token: TokenListing) {
    token.locked = true;
    self.listings.insert(token_key, &token);

    let price = token.current_offer.as_ref().unwrap().price;
    let market_keeps = self.take.multiply_balance(price);
    let others_keep = price - market_keeps;
    let receiver_id = AccountId::try_from(
        token.current_offer.as_ref().unwrap().from.to_string(),
    )
    .unwrap();
    self.ext_nft_transfer_payout(
        receiver_id,
        token_key,
        token.approval_id,
        others_keep,
    )
    .then(
        interfaces::ext_old_market::ext(env::current_account_id())
            .with_attached_deposit(NO_DEPOSIT)
            .with_static_gas(gas::PAYOUT_RESOLVE)
            .resolve_nft_payout(
                token_key.to_string(),
                token,
                others_keep.into(),
                market_keeps.into(),
            ),
    );
}
```

However, `kick_token` does not respect the lock and may result in loss of funds for both the `Marketplace` and the users.

mb-legacy-market/src/lib.rs RUST

```
pub fn kick_tokens(&mut self, token_keys: Vec<String>) {
    self.assert_owner_marketplace();
    token_keys.into_iter().for_each(|token_key| {
        let token = self.get_token(token_key.clone());
        let key: TokenKey = token_key.as_str().into();
        self.delist_internal(&key, token);
    });
}
```

The sequence of transactions provided below demonstrates a case where the `Marketplace` suffers from the excessive refund on the original `nftA` offering (block2.tx1.2), and `accountC` suffers from losing the custody (block5.tx1.2) transferred to `Marketplace` while making an offer (block4.tx1).

block1	tx1	`withdraw_offer` is called and the offer is accepted (`accountA` -`nftA`-> `accountB`). 1. `help_transfer` is called. 1-1. `accountA` owned `nftA` is locked. 1-2. `nft_transfer_payout` receipt is generated. 1-3. `resolve_nft_payout` receipt is generated,
block2	tx1	`admin` calls `kick_token` on `nftA`. 1. `nftA` is removed from `Marketplace::listing`. 2. `accountB` is refunded for `nftA`. 2-1. `token.current_offer.price` is deducted from `Marketplace` balance. 2-2. `transfer` receipt to credit `accountB` is generated.
	tx2	`nft_transfer_payout` receipt is processed. 1. `nftA` ownership is assigned to `accountB` in `MintbaseStore`.
block3	tx1	`transfer` receipt generated by `kick_token` is processed.
	tx2	`accountB` lists `nftA` on `Marketplace`.
block4	tx1	`accountC` makes offer for `nftA`.
block5	tx1	`resolve_nft_payout` receipt is processed. 1. `accountB` offer payout is resolved, `transfer` receipt is generated to pay `accountA`. 2. `nftA` listing is removed from `Marketplace`, `accountC` loses deposited funds.
block6	tx1	`transfer` receipt generated by `resolve_nft_payout` is processed.

For the provided scenario to occur, the `resolve_nft_payout` receipt must be delayed for two additional blocks and ready to be processed by block3, but not processed until block5 in order to make time for `accountB` to list `nftA`, and for `accountC` to make an offer. This delay is unlikely under the current NEAR environment, where each validator tracks all shards. However, in the future, where each validator tracks only a proportion of shards, this scenario is possible due to communication delays between nodes.

Notably, the delay of receipt processing is generally not within the control of individual users. Thus, this bug is more likely to occur by accident rather than by being performed with the intention of an attack.

Remediation

Require `kick_token` to respect the listing lock.

mb-legacy-market/src/lib.rs DIFF

```
  self.assert_owner_marketplace();
  token_keys.into_iter().for_each(|token_key| {
      let token = self.get_token(token_key.clone());
+     token.assert_not_locked();
      let key: TokenKey = token_key.as_str().into();
      self.delist_internal(&key, token);
  });
```

Patch

Resolved in 6cbc645.

OS-MTB-ADV-04 [high] | Remove Offer Does Not Respect Offer Non-Mutability

Description

The interop market enforces listings with offers that should not be tampered with, similar to the concept of locking in OS-MTB-ADV-03.

mb-interop-market/src/offer.rs RUST

```rust
pub fn buy(
    &mut self,
    nft_contract_id: AccountId,
    token_id: String,
    referrer_id: Option<AccountId>,
    affiliate_id: Option<AccountId>,
) -> Promise {
    ...
    near_assert!(
        listing.current_offer.is_none(),
        "Another offer currently executes on this listing"
    );

    // Happy path: insert offer, log event, process stuff
    let offer = Offer {
        offerer_id: env::predecessor_account_id(),
        amount: env::attached_deposit(),
        referrer_id: referrer_id.clone(),
        referral_cut,
    };
    ...
    listing.current_offer = Some(offer);
    ...
}
```

However, `remove_offer` is allowed to bypass the rule and may result in a loss of funds for market users.

mb-interop-market/src/offer.rs RUST

```rust
pub fn remove_offer(
    &mut self,
    nft_contract_id: AccountId,
    token_id: String,
) {
    // only owner is allowed to call this
    self.assert_predecessor_is_owner();

    // fetch listing
    let token_key = format!("{}<$>{}", nft_contract_id, token_id);
    let listing = self.get_listing_internal(&token_key);
    near_assert!(listing.is_some(), "Listing does not exist");
    let mut listing = listing.unwrap();
    near_assert!(
        listing.current_offer.is_some(),
        "Listing does not have an offer"
    );

    // remove offer and store
    listing.current_offer = None;
    self.listings.insert(&token_key, &listing);
}
```

The sequence of transactions provided below demonstrates a case where `accountA` may never get paid for the sale of `nftA`.

block1	tx1	buy is called (accountA -nftA-> accountB). 1. A new current_offer is added to listing. 2. execute_transfer is called. 2-1. nft_transfer_payout receipt is generated. 2-2. resolve_nft_payout_near receipt is generated.
block2	tx1	admin calls remove_offer on nftA. 1. current_offer is removed from nftA listing.
	tx2	nft_transfer_payout receipt is processed. 1. nftA ownership is assigned to accountB in MintbaseStore.
	tx3	accountA removed listing of nftA through unlist_single_nft.
block3	tx1	accountB lists nftA on Market.
block4	tx1	accountC makes offer for nftA through buy. 1. A new current_offer is added to listing. 2. execute_transfer is called. 2-1. nft_transfer_payout receipt is generated. 2-2. resolve_nft_payout_near receipt is generated.
block5	tx1	resolve_nft_payout receipt generated in block1 is processed. 1. current_offer for nftA is removed from nftA listing. 1-1. accountB is paid for this sale while accountA is never paid.

There exist many variations where the bug may manifest itself. With only the first two blocks, accountA will suffer from not receiving the payment for the nftA sale. With the remaining three blocks, it is demonstrated that while unlikely, it is possible for the sequence of listing, buying, and removing offers to be extended indefinitely. This would result in a delay of explicit transaction failure and create more difficulties in tracing back to the source of the issue.

Remediation

Mintbase expressed that the ability to forcefully remove current_offer is required to recover from potential resolve_nft_payout_near / resolve_nft_payout_ft failures.

After taking this requirement into consideration, we suggest the interop market admin monitor on-chain activities and only call remove_transfer on resolve_nft_payout_(near/ft) failures.

Patch

Guidelines for admin actions are documented in 0f5efc2.

OS-MTB-ADV-05 [high] | Potential Loss Of NFT Due To Unaccounted Fee

Description

Requiring an attachment of one yNEAR is a common method to ensure that the caller signed the transaction with a full access key. While one yNEAR is an infinitely small value, it will potentially lead to transaction failures if the caller does not have an excessive balance to fund it.

In `nft_resolve_payout_ft`, if all previous receipts regarding NFT transfers are processed successfully, `ft_transfer` with one yNEAR attached will be called to transfer `ft` to each recipient.

mb-store/src/core.rs RUST

```rust
pub fn nft_resolve_payout_ft(
    &mut self,
    token_key: String,
) -> PromiseOrValue<U128> {
    ...
    for (account, amount) in payout.drain() {
        ft_transfer(ft_contract_id.clone(), account, amount.0);
    }
    if let Some(referrer_id) = offer.referrer_id {
        ft_transfer(ft_contract_id, referrer_id, ref_earning.unwrap());
    }
    ...
}
```

mb-sdk/src/utils.rs RUST

```rust
pub fn ft_transfer(
    ft_contract_id: AccountId,
    receiver_id: AccountId,
    amount: Balance,
) -> Promise {
    crate::interfaces::ext_ft::ext(ft_contract_id)
        .with_attached_deposit(1)
        .with_static_gas(crate::constants::gas::FT_TRANSFER)
        .ft_transfer(receiver_id, amount.into(), None)
}
```

However, since there is no explicit funding for the attached yNEAR, `Marketplace` may not have the funds required, leading to the failure of `nft_resolve_payout_ft`.

Upon failure of `nft_resolve_payout_ft`, `ft_resolve_transfer` should roll back the previous payment and return all `ft` to the NFT buyer. On the other hand, `nft_transfer_payout` changes will not be rolled back, which leads to the seller losing NFTs without receiving the payment for it.

This scenario is an edge case that may only occur if the admin decides to withdraw all non-storage stake funds right before the `nft_resolve_payout_ft` receipt is processed.

The reason for the affected scenario being limited is due to NEAR distributing parts of transaction fees to contracts as developer incentives, and reception of fees for any single transaction is likely able to cover the fee amount required for `ft_transfer` to succeed.

Remediation

Withhold a part of the refunded storage fee for `listing` and use it to fund `ft_transfer`.

mb-interop-market/src/offers.rs DIFF

```
pub fn nft_resolve_payout_ft(
    &mut self,
    token_key: String,
) -> PromiseOrValue<U128> {
    ...
    for (account, amount) in payout.drain() {
        ft_transfer(ft_contract_id.clone(), account, amount.0);
    }
    if let Some(referrer_id) = offer.referrer_id {
        ft_transfer(ft_contract_id, referrer_id, ref_earning.unwrap());
    }

    self.listings.remove(&token_key);
-   self.refund_listings(&listing.nft_owner_id, 1);
+   // payout length is capped at MAX_LEN_PAYOUT_FT (10)
+   //   withholding 11 yNEAR is enough to fund ft_transfer
+   self.decrease_listings_count(&listing.nft_owner_id, 1);
+   self.refund_storage_deposit(
+       account,
+       self.listing_storage_deposit - 11u128,
+   );

    PromiseOrValue::Value(0.into())
}
```

Patch

Resolved in 13d0400 and 3eef340.

OS-MTB-ADV-06 [med] | Excessive Approval Renders Functions Unavailable

Description

NEP-178 mandates that `nft_approve`'s panic will cause `nft_revoke_all` to fail due to gas exhaust. This requirement is set for good reason, as a handful of functions may need to iterate approvals, such as how successful NFT transfers revoke all approvals.

mb-store/src/approvals.rs RUST

```rust
pub fn nft_revoke_all(&mut self, token_id: U64) -> Promise {
    ...

    let refund = token.approvals.len() as u128 * self.storage_costs.common;

    if !token.approvals.is_empty() {
        token.approvals.clear();
        self.tokens.insert(&token_idu64, &token);
        log_revoke_all(token_idu64);
    }
    Promise::new(env::predecessor_account_id()).transfer(refund)
}
```

Its inconsideration may result in users temporarily bricking their NFTs by adding too many approvals. This would become especially relevant when considering cross-contract calls that provide a static gas budget regardless of approval length, such as, `nft_transfer_payout` in `help_transfer`.

Remediation

Set a reasonable cap on the maximum amount of approvals an NFT may have. The provided code snippet below introduces a new constant `MAX_NFT_APPROVALS`, which must be set properly.

mb-store/src/approvals.rs DIFF

```diff
fn approve_internal(
    &mut self,
    token_idu64: u64,
    account_id: &AccountId,
) -> u64 {
    ...

    let approval_id = self.num_approved;
    self.num_approved += 1;

+   assert!(self.num_approved <= MAX_NFT_APPROVALS);

    token.approvals.insert(account_id.clone(), approval_id);
    self.tokens.insert(&token_idu64, &token);
    approval_id
```

Patch

Resolved in 65405c9.

OS-MTB-ADV-07 [med] | Minter Privilege Not Revocable In MintbaseStore

Description

`withdraw_minter` asserts that a transaction is signed with a full access key by calling `assert_one_yocto`. However, since it is not marked as `#[payable]`, the function does not allow the caller to attach any yNEAR to the transaction, thus making it impossible to pass the check.

Remediation

Mark `withdraw_minter` as `#[payable]`.

mb-store/src/minting.rs DIFF

```
+ #[payable]
  pub fn withdraw_minter(&mut self) {
      assert_one_yocto();
      self.revoke_minter_internal(&env::predecessor_account_id())
  }
```

Patch

Resolved in db7ef02.

OS-MTB-ADV-08 [med] | NFT State Not Restored On Transfer Failure

Description

`nft_resolve_transfer` should fully rollback the NFT state if the following `nft_on_transfer` call fails or returns true.

The current implementation of `nft_resolve_transfer` does not properly restore `split_owners` and `approvals` to their original state. Both fields are cleared regardless of the transfer result.

Remediation

Restore `token.split_owners` and `token.approvals` if the transfer fails.

mb-store/src/core.rs DIFF

```
#[payable]
pub fn nft_transfer_call(
    &mut self,
    receiver_id: AccountId,
    token_id: U64,
    approval_id: Option<u64>,
    msg: String,
) -> Promise {
    ...

    // prevent race condition, temporarily lock-replace owner
    let owner_id = AccountId::new_unchecked(token.owner_id.to_string());
+   let approvals = token.approvals.clone();
+   let split_owners = token.split_owners.clone();
    self.transfer_internal(&mut token, receiver_id.clone(), true);
    log_nft_transfer(
        &receiver_id,
        token_id_u64,
        &None,
        owner_id.to_string(),
    );
    self.lock_token(&mut token);

    ext_nft_on_transfer::ext(receiver_id.clone())
        .with_static_gas(gas::NFT_TRANSFER_CALL)
        .nft_on_transfer(pred, owner_id.clone(), token_id, msg)
        .then(
            store_self::ext(env::current_account_id())
                .with_static_gas(gas::NFT_TRANSFER_CALL)
                .nft_resolve_transfer(
                    owner_id,
                    receiver_id,
                    token_id.0.to_string(),
                    None,
+                   approvals,
+                   split_owners,
                ),
        )
}

#[private]
pub fn nft_resolve_transfer(
    &mut self,
    owner_id: AccountId,
    receiver_id: AccountId,
    token_id: String,
    // NOTE: might borsh::maybestd::collections::HashMap be more appropriate?
    approved_account_ids: Option<HashMap<AccountId, u64>>,
+   approvals: HashMap<AccountId, u64>,
+   split_owners: Option<SplitOwners>,
```

```
) -> bool {
    let l = format!(
-       "owner_id={} receiver_id={} token_id={} split_owners={:?} pred={}",
+       "owner_id={} receiver_id={} token_id={} approved_ids={:?} approvals={:?} split_owners={:?} pred={}",
        owner_id,
        receiver_id,
        token_id,
        approved_account_ids,
+       approvals,
+       split_owners,
        env::predecessor_account_id()
    );
    ...
    if !must_revert {
        true
    } else {
        self.transfer_internal(&mut token, owner_id.clone(), true);
+       token.approvals = approvals;
+       token.split_owners = split_owners;
        log_nft_transfer(
            &owner_id,
            token_id_u64,
            &None,
            receiver_id.to_string(),
        );
        false
    }
}
```

Patch

Resolved in 189b0bb.

OS-MTB-ADV-09 [med] | Storage Fee Uncollected When Listing NFTs

Description

`nft_on_approve` adds NFTs to `listings` and increases storage usage. Since storage fees are not collected from the caller, it is possible for an attacker to lock up all funds in `Marketplace` by repeatedly approving NFTs to it.

mb-legacy-market/src/listings.rs RUST

```rust
pub fn nft_on_approve(
    &mut self,
    token_id: U64,
    owner_id: AccountId,
    approval_id: u64,
    msg: String, // try to parse into saleArgs
) {
    ...
    let token = self.listing_insert_internal(
        token_id,
        U64(approval_id),
        &owner_id,
        &sale_args,
    );
    ...
}

pub(crate) fn listing_insert_internal(
    &mut self,
    token_id: U64,
    approval_id: U64,
    owner_id: &AccountId,
    sale_args: &mb_sdk::data::market_v1::SaleArgs,
) -> TokenListing {
    let approval_id: u64 = approval_id.into();
    // Create the tokens. Skip any tokens that are already listed.
    let key = TokenKey {
        token_id: token_id.0,
        account_id: env::predecessor_account_id().to_string(),
    };
    let token = TokenListing::new(
        owner_id.clone(),
        env::predecessor_account_id(),
        token_id.into(),
        approval_id,
        sale_args.autotransfer,
        sale_args.price,
    );
    match self.listings.get(&key) {
        None => {
            self.listings.insert(&key, &token);
        }
        Some(old_token) => {
            // token has been relisted, handle old token data and reinsert.
            self.delist_internal(&key, old_token);
            self.listings.insert(&key, &token);
        }
    }
    token
}
```

Remediation

Require approvals to properly fund potential storage usage.

Patch

Mintbase acknowledged this issue and decided to keep the original implementation for NEP compatibility reasons. `Marketplace` will be refunded by Mintbase to account for additional locked storage fees if necessary.

OS-MTB-ADV-10 [med] | Updating Token Owner Does Not Charge Caller

Description

When a token is transferred to an account without any NFTs, the receiving account will be added to `tokens_per_owner`.

mb-store/src/lib.rs RUST

```rust
fn update_tokens_per_owner(
    ...
) {
    ...
    if let Some(to) = to {
        let mut new_owner_owned_set = self.get_or_make_new_owner_set(&to);
        new_owner_owned_set.insert(&token_id);
        self.tokens_per_owner.insert(&to, &new_owner_owned_set);
    }
}
```

The storage cost for this is not accounted for in `update_tokens_per_owner` and its callers, thus allowing attackers to lock up funds by repeatedly transferring NFTs to new accounts.

Remediation

Require NFT minters to fully sponsor storage fees for potential listings.

mb-store/src/minting.rs DIFF

```
  fn storage_cost_to_mint(
      ...
  ) -> near_sdk::Balance {
-     // create an entry in tokens_per_owner
-     self.storage_costs.common
-     // create a metadata record
-     + metadata_storage as u128 * self.storage_costs.storage_price_per_byte
+     // create a metadata record
+     metadata_storage as u128 * self.storage_costs.storage_price_per_byte
      // create a royalty record
      + num_royalties as u128 * self.storage_costs.common
      // create n tokens each with splits stored on-token
-     + num_tokens as u128 * (self.storage_costs.token + num_splits as u128 * self.storage_costs.common)
+     + num_tokens as u128 * (
+         // token base storage
+         self.storage_costs.token
+         // dynamic split storage
+         + num_splits as u128 * self.storage_costs.common
+         // create an entry in tokens_per_owner
+         + self.storage_costs.common
+     )
  }
```

Patch

Resolved in a434371.

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-MTB-SUG-00	Avoid redundant state checks in `#[init]` functions to reduce gas usage.
OS-MTB-SUG-01	There are contradictory aspects in the documentation and code.
OS-MTB-SUG-02	Avoid manual calculation of storage fees as they are fragile and are a common source of issues that lead to a denial of service.

OS-MTB-SUG-00 | Avoid Redundant State Checks In Init Functions

Description

The `#[init]` macro includes an implicit check for `env::state_exists()` to be `false`, as demonstrated in the provided code snippet below. Repeating the check in the function body unnecessarily uses gas.

near-sdk-macros/src/core_impl/code_generator/impl_item_method_info.rs RUST

```
fn init_method_wrapper(
    method_info: &ImplItemMethodInfo,
    check_state: bool,
) -> Result<TokenStream2, syn::Error> {
    let ImplItemMethodInfo { attr_signature_info, struct_type, .. } = method_info;
    let arg_list = attr_signature_info.arg_list();
    let AttrSigInfo { ident, returns, is_handles_result, .. } = attr_signature_info;
    let state_check = if check_state {
        quote! {
            if near_sdk::env::state_exists() {
                near_sdk::env::panic_str("The contract has already been initialized");
            }
        }
    } else {
        quote! {}
    };
    ...
}
```

Remediation

Remove `near_assert!(!env::state_exists(), ...)` from `#[init]` functions.

mb-legacy-market/src/lib.rs DIFF

```
pub fn new(metadata: NFTContractMetadata, owner_id: AccountId) -> Self {
-   near_assert!(
-       !env::state_exists(),
-       "This store is already initialized!"
-   );
    let mut minters = UnorderedSet::new(b"a".to_vec());
    minters.insert(&owner_id);
    ...
}
```

mb-legacy-market/src/lib.rs DIFF

```
#[init]
pub fn new(init_allowlist: Vec<AccountId>) -> Self {
-   near_assert!(!env::state_exists(), "Already initialized");

    let mut allowlist = UnorderedSet::new(b"a".to_vec());
    ...
}
```

Patch

Resolved in fd8f281, 3e16fd3 and 117c511

OS-MTB-SUG-01 | Contradictory Documentation And Code

Description

1. Rules regarding offer validation in the legacy market:

The documentation on offer-making rules does not match its implementation.

mb-legacy-market/src/offers.rs RUST

```
impl Marketplace {
    /// Make an `Offer` for `Token`. `Offer`s may be created beneath the `Token`'s
    /// `asking_price`, but not beneath the `current_offer`'s price, unless the
    /// current offer has timed out.
    ///
    /// The `price` argument MUST be >= `env::attached_deposit` on this function.
    ...
}
```

mb-legacy-market/src/offers.rs RUST

```
pub fn make_offer(
    &mut self,
    token_key: Vec<String>,
    price: Vec<U128>,
    timeout: Vec<TimeUnit>,
) {
    ...
    let mut total: Balance = 0;
    let token_offers = token_key
        .into_iter()
        .zip(price.into_iter())
        .zip(timeout.into_iter())
        .map(|((token_key, price), timeout)| {
            total += price.0;
            ...
            self.try_make_offer(&mut listing, offer.clone());
            ...
        })
        .collect::<Vec<_>>();
    near_assert!(
        total == env::attached_deposit(),
        "Summed prices must match the attached deposit",
    );
    ...
}

fn try_make_offer(&mut self, token: &mut TokenListing, offer: TokenOffer) {
    ...
    near_assert!(
        offer.price >= token.asking_price.into(),
        "Cannot set offer below ask"
    );
    ...
}
```

2. Minting limits in `MintbaseStore`:

The documentation on minting limits does not match its implementation.

mb-store/src/minting.rs RUST

```rust
impl MintbaseStore {
    ...
    /// - Because of logging limits, this method may mint at most 99 tokens per call.
    /// - 1.0 >= `royalty_f` >= 0.0. `royalty_f` is ignored if `royalty` is `None`.
    /// - If a `royalty` is provided, percentages **must** be non-negative and add to one.
    /// - The maximum length of the royalty mapping is 50.
    ///
    /// This method is the most significant increase of storage costs on this
    /// contract. Minters are expected to manage their own storage costs.
    ...
}
```

mb-store/src/minting.rs RUST

```rust
#[payable]
pub fn nft_batch_mint(
    &mut self,
    owner_id: AccountId,
    #[allow(unused_mut)] // cargo complains, but it's required
    mut metadata: TokenMetadata,
    num_to_mint: u64,
    royalty_args: Option<RoyaltyArgs>,
    split_owners: Option<SplitBetweenUnparsed>,
) -> PromiseOrValue<()> {
    near_assert!(num_to_mint > 0, "No tokens to mint");
    near_assert!(
        num_to_mint <= 125,
        "Cannot mint more than 125 tokens due to gas limits"
    ); // upper gas limit
    ...
    near_assert!(
        roy_len + split_len <= MAX_LEN_PAYOUT,
        "Number of payout addresses may not exceed {}",
        MAX_LEN_PAYOUT
    );
    ...
}
```

Remediation

Update the documentation and code to avoid confusion.

1. Offer-making rules:

mb-legacy-market/src/offers.rs DIFF

```
impl Marketplace {
-   /// Make an `Offer` for `Token`. `Offer`s may be created beneath the `Token`'s
-   /// `asking_price`, but not beneath the `current_offer`'s price, unless the
-   /// current offer has timed out.
-   ///
-   /// The `price` argument MUST be >= `env::attached_deposit` on this function.
+   /// Make an `Offer` for `Token`. If the token is listed as simple sale (aka
+   /// "buy now", `autotransfer` is `true`), the offer price may not be below the
+   /// asking price, If the token is listed as rolling auction (`autotransfer` is
+   /// `false`), you may place an offer below the asking price.
+   ///
+   /// The `price` argument MUST be <= `env::attached_deposit` on this function.
    ...
}
```

mb-legacy-market/src/offers.rs DIFF

```
fn try_make_offer(&mut self, token: &mut TokenListing, offer: TokenOffer) {
    ...
    near_assert!(
-       offer.price >= token.asking_price.into(),
-       "Cannot set offer below ask"
+       !token.autotransfer || offer.price >= token.asking_price.into(),
+       "Cannot set offer below ask for simple sales"
    );
    ...
}
```

2. Minting limits:

mb-store/src/minting.rs DIFF

```
impl MintbaseStore {
    ...
-   /// - Because of logging limits, this method may mint at most 99 tokens per call.
+   /// - Because of logging limits, this method may mint at most 125 tokens per call.
    /// - 1.0 >= `royalty_f` >= 0.0. `royalty_f` is ignored if `royalty` is `None`.
    /// - If a `royalty` is provided, percentages **must** be non-negative and add to one.
-   /// - The maximum length of the royalty mapping is 50.
+   /// - The maximum length of the royalty mapping is MAX_LEN_PAYOUT - 1.
    ///
    /// This method is the most significant increase of storage costs on this
    /// contract. Minters are expected to manage their own storage costs.
    ...
}
```

Patch

Resolved in 0c1249e and c86ba6a.

OS-MTB-SUG-02 | Avoid Manual Calculation Of Storage Fees

Description

In the NEAR ecosystem, locked storage fees are sponsored by the contract rather than the callers. This creates a unique attack surface where if attackers have the ability to increase storage usage of contracts without paying for it, contract funds may eventually become locked.

To prevent this kind of attack, contracts must carefully consider potential changes in storage usage and charge users accordingly through attached assets.

Mintbase manages storage fees through pre-calculated structure sizes, admin-assigned storage fees per byte, and tracking the addition and removal of storage manually.

Miscalculated and unaccounted storage usage and potential future issues are listed below.

- Storage fees are not guaranteed to stay constant. Admins are required to promptly update contracts or set `storage_price_per_byte` on any future NEAR environment changes.
- Tracking all possible storage consumption changes is difficult, and any future updates to the contract code would require developers to re-assess storage usage logic.

Remediation

We recommend NEAR developers call `env::storage_usage` twice, once upon entering the contract, and once when leaving the contract. Then, take the difference as storage delta and multiply this value by `env::storage_byte_cost` to obtain the storage fee that the caller must pay.

Patch

Mintbase acknowledged our recommendation and decided to keep the original implementation to ensure API fees are constant and predictable for SDK.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority or access control validation • Improperly designed economic incentives leading to loss of funds
High	Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities that could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.