Guvenkaya® The Bedrock of Security

# DeFiShards

## NEAR Rust Smart Contract Security Assessment

Lead Security Engineer: Michal Bajor
Date of Engagement: 9th September 2024 - 27th September 2024
Visit: www.guvenkaya.co

# Contents

# About Us

Guvenkaya is a security research firm specializing in Rust security, Web3 security of Rust-based protocols, and Web2 security. With our expertise, we provide both security auditing services and custom security solutions

# About DeFiShards

DeFiShards is a decentralized investment platform built on NEAR Protocol that leverages NFTs to provide access to various DeFi products. Users can mint NFTs representing ownership of different assets or financial products.

# Audit Results

Guvenkaya conducted a security assessment of the DeFiShards smart contracts from 9th Septembet 2024 to 27th September 2024. During this engagement, a total of 16 findings were reported. 6 of the findings were critical, 1 high, 2 medium, and the remaining were either low or informational severity. The DeFiShards team has fixed all the major issues

## Project Scope

Lines of Code Reviewed: 2523

FT Smart Contract

| File name | Link |
|---|---|
| Lib | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/ft/src/lib.rs |

Launchpad Smart Contract

| File name | Link |
|---|---|
| Lib | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/launchpad/src/lib.rs |

Marketplace Smart Contract

| File name | Link |
|---|---|
| Lib | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/marketplace/src/lib.rs |
| NFT Callbacks | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/marketplace/src/nft_callbacks.rs |
| External | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/marketplace/src/external.rs |

NFT Smart Contract

| File name | Link |
|---|---|
| Lib | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/nft/src/lib.rs |
| FT Balances | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/nft/src/ft_balances.rs |
| External | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/nft/src/external.rs |

Vault Smart Contract

| File name | Link |
|---|---|
| Lib | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/vault/src/lib.rs |

Dapp

| File name | Link |
|---|---|
| Config | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/dapp/src/config.js |
| Wallet Selector | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/dapp/src/wallets/wallet-selector.js |
| Web3 Wallet | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/dapp/src/wallets/web3-wallet.ts |
| Cards | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/dapp/src/components/cards.js |
| Navigation | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/dapp/src/components/navigation.js |
| VM Component | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/dapp/src/components/vm-component.js |
| Page | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/dapp/src/app/page.js |

| File name | Link |
|---|---|
| Layout | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/dapp/src/app/layout.js |
| Page (Hello NEAR) | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/dapp/src/app/hello-near/page.js |
| Page (Hello Components) | https://github.com/kdbvier/Near-Marketplace/blob/b75412f639aa792cc8268733d99ebbfe9393c68a/dapp/src/app/hello-components/page.js |

## Out of Scope

The audit will include, but is not limited to, reviewing the code for security vulnerabilities, coding practices, and architecture. The audit does not include a review of the dependencies. The audit does not include reviewing of the auction functionality since it is work in progress. Issues related to the audit functionality (**GUV-6** and **GUV-7**) are found as part of the main audit and will be resolved in the next release as soon as the auction is ready

## Timeline

| Start of the audit | Draft report | Final report |
| --- | --- | --- |
| 9th September 2024 | 27th September 2024 | 21st October 2024 |

# Methodology

| |
|---|
| RESEARCH INTO PROJECT ARCHITECTURE |
| PREPARING ATTACK VECTORS |
| SETTING UP AN ENVIRONMENT |
| MANUAL CODE REVIEW OF THE CODE |
| ASSESSMENT OF RUST SECURITY ISSUES |
| ASSESSMENT OF NEAR SECURITY ISSUES |
| ASSESSMENT OF ARITHMETIC ISSUES |
| BUSINESS LOGIC VULNERABILITY ASSESSMENT |
| ONCHAIN TESTING USING NEAR WORKSPACES |
| BEST PRACTICES AND CODE QUALITY |
| CHECKING FOR CODE REFACTORING/SIMPLIFICATION POSSIBILITIES |
| ARCHITECTURE IMPROVEMENT SUGGESTIONS |
| PREPARING POCS AND/OR TESTS FOR EACH CRITICAL/HIGH/MEDIUM ISSUES |

# Severity Breakdown

## 01. Likelihood Ratings

**Likely:** The vulnerability is easily discoverable and not overly complex to exploit.
**Possible:** The vulnerability presents some challenges either in discovery or in the complexity of the attack.
**Rare:** The vulnerability is either very difcult to discover or complex to exploit, or both.
This matrix provides a nuanced view, taking into account both the ease of discovering a vulnerability and the complexity involved in exploiting it.

## 02. Impact

**Severe:** The vulnerability is easily discoverable and not overly complex to exploit.
**Moderate:** The vulnerability presents some challenges either in discovery or in the complexity of the attack.
**Negligible:** The vulnerability is either very difcult to discover or complex to exploit, or both.

## 03. Severity Ratings

**Critical:** Assigned to vulnerabilities with severe impact and a likely likelihood of exploitation.
**High:** For vulnerabilities with either severe impact but only a possible likelihood, or moderate impact with a likely likelihood.
**Medium:** Used for vulnerabilities with severe impact but a rare likelihood, moderate impact with a possible likelihood, or negligible impact with a likely likelihood.
**Low:** For vulnerabilities with moderate impact and rare likelihood, or negligible impact with a possible likelihood.
**Informational:** The lowest severity rating, typically for vulnerabilities with negligible impact and a rare likelihood of exploitation.

CRITICAL HIGH MEDIUM Low Informational

## Likelihood Matrix:

| Attack Complexity \ Discovery Ease | Obvious | Concealed | Hidden |
|---|---|---|---|
| Complex | Possible | Rare | Rare |
| Moderate | Likely | Possible | Rare |
| Straightforward | Likely | Possible | Possible |

## Likelihood/Impact Matrix:

| Likelihood \ Impact | Severe | Moderate | Negligible |
|---|---|---|---|
| Likely | **CRITICAL** | **HIGH** | **MEDIUM** |
| Possible | **HIGH** | **MEDIUM** | Low |
| Rare | **MEDIUM** | Low | Informational |

# Findings Summary

**01. Remediation Complexity:** This measures how difcult it is to fx the vulnerability once it has been identifed.

**Simple:** Patches or fixes are readily available and easily implemented.

**Moderate:** Requires some time and resources to remediate, but well within the capabilities of most organizations.

**Difficult:** Remediation requires significant resources, specialized skills, or substantial changes to systems or architecture.

**02. Status:** This measures how difcult it is to fx the vulnerability once it has been identifed.

**Not Fixed:** Indicates that the vulnerability has been identifed but no remedial action has been taken yet. This status is crucial for newly discovered vulnerabilities or those awaiting prioritization.

**Fixed:** This status is applied when the vulnerability has been successfully remediated. It implies that appropriate measures (like patching, confguration changes, or architectural modifcations) have been implemented to resolve the issue.

**Acknowledged:** This status is used for vulnerabilities that have been recognized, but for various reasons (such as risk acceptance, cost, or other business decisions), have not been fxed. It indicates that the risk posed by the vulnerability is known and has been consciously accepted.

| Finding | Impact | Likelihood | Severity | Remediation Complexity | Remediation Status |
|---|---|---|---|---|---|
| GUV-1: Irrecoverable Denial-of-Service Condition In Vault Contract | Severe | Likely | CRITICAL | Moderate | Fixed |
| GUV-2: Unrestricted Native Deposits Lead To An Unoperational Vault Contract Due To Invalid Accounting | Severe | Likely | CRITICAL | Simple | Fixed |
| GUV-3: Unrestricted NFT Minting Leads To A Denial-of-Service Condition | Severe | Likely | CRITICAL | Simple | Fixed |
| GUV-4: Certain Token Ids Disrupt The Protocol | Severe | Likely | CRITICAL | Simple | Fixed |
| GUV-5: Multiple Issues With The Launchpad Contract | Severe | Likely | CRITICAL | Simple | Fixed |
| GUV-6: Auction Bypass | Severe | Likely | CRITICAL | Simple | Future Release |
| GUV-7: Possibility Of Deleting Market Data For a Live Auction | Severe | Possible | HIGH | Moderate | Future Release |
| GUV-8: Jeopardizing Marketplace Process Via Manual NFT Transfer | Negligible | Likely | MEDIUM | Simple | Acknowledged |
| GUV-9: Lack of Promise Callback Usage | Moderate | Possible | MEDIUM | Moderate | Partially Fixed |
| GUV-10: Missing assert_one_yocto | Negligible | Possible | Low | Simple | Fixed |
| GUV-11: Suboptimal Implementation Of Storage Staking Mechanism | Negligible | Possible | Low | Moderate | Fixed |
| GUV-12: Possible Precision Loss | Negligible | Possible | Low | Moderate | Partially Fixed |
| GUV-13: Possible Token Lock In The Vault | Negligible | Possible | Low | Moderate | Fixed |

| Finding | Impact | Likelihood | Severity | Remediation Complexity | Remediation Status |
|---|---|---|---|---|---|
| GUV-14: Hardcoded Storage Cost | Negligible | Possible | Low | Simple | Fixed |
| GUV-15: Unverified Calculation Parameters | Negligible | Rare | Informational | Simple | Fixed |
| GUV-16: Miscellaneous Notes | Negligible | Rare | Informational | Moderate | Acknowledged |

# Findings Details

## GUV-1 Irrecoverable Denial-of-Service Condition In Vault Contract - Critical

The *vault* contract is responsbile for holding the assets which are supposed to be transferred whenever an *nft* contract (*vault*'s owner) calls the *withdraw* function. The *vault* contract is supposed to accept multiple fungible tokens as part of*ft_transfer_call* execution flow. However, no whitelist is implemented, making it possible for anyone to call the*ft_on_trasfer* function. As such, the internal *owned_fts* Map can be filled with arbitrarily chosen data. This Map is then looped over inside the *withdraw* function, and a *Promise* is created for each entry. As the max gas that can be attached to the call is limited, if there are enough entries in the *owned_fts* mapping, the*withdraw* functionality will encounter an *OutOfGas* error which will stop the execution with an error. Furthermore, no *Promise* will be scheduled. Such a scenario renders the *vault* contract useless and is irrecoverable.

vault/src/lib.rs

```
fn ft_on_transfer(
    &mut self,
    sender_id: AccountId,
    amount: U128,
) -> U128 {
    // get the contract ID which is the predecessor
    let ft_contract_id = env::predecessor_account_id();
    if ft_contract_id == self.ft_contract.clone().unwrap() {
        //get the signer which is the person who initiated the transaction
        let signer_id = env::signer_account_id();
```

vault/src/lib.rs

```
            //make sure that the signer isn't the predecessor. This is so that we're sure
            //this was called via a cross-contract call
            assert_ne!(
                ft_contract_id,
                signer_id,
                "ft_on_transfer should only be called via cross-contract call"
            );
            self.amount = self.amount + amount.0;
        } else {
            let prev_amount = self.owned_fts.get(&ft_contract_id).unwrap_or(0);
            let new_amount = prev_amount + amount.0;
            self.owned_fts.insert(&ft_contract_id, &new_amount);
        }
        U128(0)
    }
```

**POC**

https://gist.github.com/michaelbajor/5a2f9a0696226008aab184b8c5b07bcc

**PROPOSED SOLUTION**

It is recommended to implement a whitelist inside the *vault*contract so that it accepts the deposits only from recognized tokens. Before an airdrop is expected to happen, an administrator should add the token's *AccountId* to the whitelist. Additionally, a precausions should be in-place for cases where number of legitimate tokens will also exceed the gas limit. A solution for that might be a separate withdrawal function designed to transfer fungible tokens that could be called multiple times, should the gas limit prevent an execution in a single call.

**REMEDIATION - FIXED**

The team has implemented an access control mechanism that would allow only an AccountId associated with the `owner` role to deposit tokens into the smart contract. Additionally, the assertion is implemented that only at most three different fungible tokens can be airdropped to the vault.

## GUV-2 Unrestricted Native Deposits Lead To An Unoperational Vault Contract Due To Invalid Accounting - Critical

The *vault* contract is intended to operate primarily either on one general FT token or on native NEAR tokens, but never both. However, the variable used to track the amount of tokens submitted to the *vault* contract is the same, regardless of the type of tokens *vault* contract is configured to operate on. When *withdraw* function is used to distribute the tokens and delete the *vault* contract, it will either schedule a *Promise* with *ft_transfer* call or a *Promise* with native tranfser depending on the configuration. However, the *deposit_near* function used to depoist native tokens is not protected - anyone can call it at any time regardless of the *vault* configuration. As such, if the *vault* is configured to work with FT tokens only, a malicious user can call *deposit_near* function to modify the internal *amount* value making the *vault* contract think it owns more FT tokens than it actually does. In this scenario, during the *withdraw* execution, the *vault* contract will schedule an *ft_transfer* execution with invalid amounts. As a consequence, such a token tranfser might fail leaving some, or all, FT tokens in the *vault*'s balance. However, the *vault* contract also deletes itself after scheduling a tranfser *Promise*s, making those FT tokens irrecoverably lost.

vault/src/lib.rs

```
#[payable]
pub fn deposit_near(&mut self) {
    let attached_amount = env::attached_deposit();
    self.amount = attached_amount.as_yoctonear();
}
```

**POC**

https://gist.github.com/michaelbajor/2573e38fe1ba6a471ee364db0e50c0e0

**PROPOSED SOLUTION**

It is recommended to implement a verification mechanism in the*deposit_near* function that will allow the function to execute only if the *vault* contract is not configured to work with FT token.

**REMEDIATION - FIXED**

The team has resolved the issue by allowing *deposit_near* function to execute only if the contract is configured to work with FT token.

## GUV-3 Unrestricted NFT Minting Leads To A Denial-of-Service Condition- Critical

The *nft* contract implements an *nft_mint* function used to create new tokens within the contract. This function does not implement any authorization - anyone can call it to successfuly create NFTs. The comment specifies that the internal nft minting function will make sure that the *predecessor_account_id* is equal to the owner, but that is not the case. During the discussion about unrestricted minting, DefiShards team determined that a public*nft_mint* functionality is the desired configuration. Such design, however, leads to a possibile Denial-of-Service vulnerabilities, as it was indicated that in most cases the total supply for a given NFT will be around 10. Publicly callable*nft_mint* function would allow malicious users to quickly take control over the NFTs making it impossible for intended users to use the contract.

**PROPOSED SOLUTION**

Add a verification mechanism that would permit only owner to execute the *nft_mint* function

**REMEDIATION - FIXED**

Issues of front-running-like vulnerabilites were addressed by implementing a Merkle Tree based verification mechanism in cases the minting should be restricted only to specific users. The Access Control itself, i.e. Merkle Tree along with proof calculation will be implemented in the off-chain component.

## GUV-4 Certain Token Ids Disrupt The Protocol- Critical

The *nft_mint* function from the *nft* contract is responsible for minting the NFT with the provided token ID. However, it also deploys the vault contract to the *AccountId* constructed from the*token_id* and the *nft* contract's *AccountId* as own subaccount. The *TokenId* type is an alias for *String* type making it possible to provide any sequence of characters as a *TokenId*. Because of the created subaccount, the only requirement this ID needs to fulfill is that it needs to create a valid *AccountId* when concatenated with the *nft* contract's own *AccountId*. However, it is possible to provide a sequence of characters that will result in a valid*AccountId* while also making it impossible to create. For example, if the caller would provie the "one.two" string as a new token ID, then the constructed *AccountId* would be one.two.*nft-account-id*, which is a valid AccountId. However, that Account cannot be created by the *nft* contract, because it has two levels of domains instead of one. Hence, the vault contract will not be deployed, while the NFT itself will be created, and the callback will execute - the balance transfers to the owner will happen.

**POC**

https://gist.github.com/michaelbajor/4607519c4eda70c0129ffcf4237d085a

**PROPOSED SOLUTION**

It is recommended to construct the new token IDs internally, instead of relying on caller to provide a valid *TokenId*.

**REMEDIATION - FIXED**

The issue was fixed by internally keeping track of a most recently created NFT index and using it as a subaccount's name.

## GUV-5 Multiple Issues With The Launchpad Contract - Critical

The *launch* function defined in the *launchpad* contract can be called by anyone in order to create the *nft* contract. The function is marked as *payable*, but does not verify the attached deposit. As a consequence, caller can simply not attach any deposit and the launchpad contract will transfer own balance to the newly created NFT contract, draining the launchpad free balance. Additionally, even if the caller provides balance, the actual contract deployment can fail. However, the event related to launching will still be emitted and caller will loose the attached deposit.

### POC

https://gist.github.com/michaelbajor/d03404e219376ddaaaa91a436b8de73a

### PROPOSED SOLUTION

It is recommended to verify the attached deposit and make sure that the *launch* function caller provided enough depoist to cover all fees associated with launching new *nft* contract. Additionally, the caller-provided NFT metadata must be validated, to make sure that the created subaccount can be created by a *launchpad* contract. Furthermore, implementing a callback that would check if the execution was succesful is highly desired. That way the attached deposit could be returned to the user in case it was not and the appropriate event can be emitted only when the successful execution took place.

### REMEDIATION - FIXED

The team has fixed the balance draining vulnerability. Technically, it is still possible to provide an NFT symbol that would result in a failing contract deployment, however, the `launch` function was modified to be only callable by an AccountId associated with the `admin` role.

## GUV-6 Auction Bypass - Critical

The *marketplace* contract defines two ways of selling the NFTs - an auction or direct buy for a set price. It was observed that the*buy* function does not verify whether the *MarketData* associated with the given listing is related to auction or not. As a consequence, the ongoing auction can be interrupted by another user simply calling the *buy* function. The result of such action is that *buy* caller will receive the NFT for the original ask price (the minimum first bidder had to pay to participate in the auction). However, completeing the purchase in any way also deletes all data associated with the listing and this in turn results in locking the bidders funds within the *marketplace* contract as they can not cancel their bids without the associated *MarketData*.

marketplace/src/lib.rs

```
    #[payable]
    pub fn buy(&mut self, nft_contract_id: AccountId, token_id: TokenId) {
        let contract_and_token_id = format!("{}{}{}", &nft_contract_id, DELIMETER,
token_id);
        let market_data = self
            .market
            .get(&contract_and_token_id)
            .expect("DS: Market data doesn't exist");
        let buyer_id = env::predecessor_account_id();
        assert_ne!(
            buyer_id, market_data.owner_id,
            "DS: Cannot buy your own sale"
        );
        assert_eq!(
            env::attached_deposit().as_yoctonear(),
            market_data.price,
            "DS: Insufficient Balance"
        );
```

marketplace/src/lib.rs

```
            self.internal_process_purchase(
                nft_contract_id.into(),
                token_id,
                buyer_id,
                env::attached_deposit().as_yoctonear(),
            );
        }
```

**POC**

https://gist.github.com/michaelbajor/1938d7a19258acca6780c2994e440dc1

**PROPOSED SOLUTION**

It is recommended to implement a verification mechanism in the *buy* function that will prevent execution if the *MarketData* in question is related to the auction.

**REMEDIATION - FUTURE RELEASE**

The team has stated that, at the time of writing this report, the auction functionality is not going to be supported, but the vulnerability will be fixed once auctions are launched.

## GUV-7 Possibility Of Deleting Market Data For a Live Auction - High

One of the possible types of listings within the *marketplace*contract is an auction. In an auction, the bidders naturally are transferring increasing amounts of tokens in subsequent bids and when seller decides to accept the current highest one, the NFT transfer will happen and non-winning bidders will have their deposits returned. However, it was observed that the original NFT seller, i.e. user who created the listing, and the *marketplace* owner can call the *delete_market_data* function at any time. The *delete_market_data* function is responsible for deleteing all data related to the given listing. Consequently, all data related to the already submitted bids is lost and bidders cannot cancel their bids in order to get their tokens back.

marketplace/src/lib.rs

```
    #[payable]
    pub fn delete_market_data(&mut self, nft_contract_id: AccountId, token_id: TokenId)
{
        assert_one_yocto();
        let contract_and_token_id = format!("{}{}{}", nft_contract_id, DELIMETER,
token_id);
        let current_time: u64 = env::block_timestamp();
        let market_data = self
            .market
            .get(&contract_and_token_id)
            .expect("DS: Market data does not exist");
        assert!(
            [market_data.owner_id.clone(), self.owner_id.clone()]
                .contains(&env::predecessor_account_id()),
            "DS: Seller or owner only"
        );
```

marketplace/src/lib.rs

```
        if market_data.is_auction.is_some() && env::predecessor_account_id() ==
self.owner_id {
            assert!(
                current_time >= market_data.ended_at.unwrap(),
                "DS: Auction has not ended yet"
            );
        }
       self.internal_delete_market_data(&nft_contract_id, &token_id);

        env::log_str(
            &json!({
                "type": "delete_market_data",
                "params": {
                    "owner_id": market_data.owner_id,
                    "nft_contract_id": nft_contract_id,
                    "token_id": token_id,
                }
            })
            .to_string(),
        );
    }
```

**POC**

https://gist.github.com/michaelbajor/250d556cf89ccb44810f2ed802b87d34

**PROPOSED SOLUTION**

It is recommended to prevent a scenario where data related to live auction is deleted. This can be achieved by either preventing deletion altogether when there are already submitted bids or returning deposits to all of the bidders and then deleting the *MarketData*.

**REMEDIATION - FUTURE RELEASE**

The team has stated that, at the time of writing this report, the auction functionality is not going to be supported, but the vulnerability will be fixed once auctions are launched.

## GUV-8 Jeopardizing Marketplace Process Via Manual NFT Transfer - Medium

The *marketplace* contract uses the *nft_on_approve* function to create a listing, and subsequently offer the NFT for sale. The *nft_on_approve* function is part of the NFT standard and is called by the *nft* contract when extending the approval to a third party, in this case - *marketplace* contract. However, the approval mechanism by itself does not freeze the NFT, making it possible for the owner, or any other approved party, to transfer the NFT at any time. As a consequence, the *marketplace* contract will be collecting bids for the NFT that it cannot physically transfer. In such a scenario, the auction completion, will result in an error. Although, no tokens are locked, the whole marketplace flow is disrupted.

marketplace/src/nft_callbacks.rs

```
fn nft_on_approve(
        &mut self,
        token_id: TokenId,
        owner_id: AccountId,
        approval_id: u64,
        msg: String,
    ) {
        // enforce cross contract call and owner_id is signer
        let nft_contract_id = env::predecessor_account_id();
        let signer_id = env::signer_account_id();
        assert_ne!(
            env::current_account_id(),
            nft_contract_id,
            "DS: nft_on_approve should only be called via cross-contract call"
        );
        assert_eq!(owner_id, signer_id, "DS: owner_id should be signer_id");
```

marketplace/src/nft_callbacks.rs

```
        assert!(
            self.approved_nft_contract_ids.contains(&nft_contract_id),
            "DS: nft_contract_id is not approved"
        );
        let MarketArgs {
            price,
            started_at,
            ended_at,
            is_auction,
            end_price,
        } = near_sdk::serde_json::from_str(&msg).expect("Not valid MarketArgs");
        assert!(price.is_some(), "DS: price not specified");
        let storage_amount = self.storage_minimum_balance().0;
        let owner_paid_storage = self.storage_deposits.get(&signer_id).unwrap_or(0);
        let signer_storage_required =
            (self.get_supply_by_owner_id(signer_id).0 + 1) as u128 * storage_amount;
        if owner_paid_storage < signer_storage_required {
            let notif = format!(
                "Insufficient storage paid: {}, for {} sales at {} rate of per sale",
                owner_paid_storage,
                signer_storage_required / storage_amount,
                storage_amount
            );
            env::log_str(&notif);
            return;
        }
```

marketplace/src/nft_callbacks.rs

```
            self.internal_add_market_data(
                owner_id,
                approval_id,
                nft_contract_id,
                token_id,
                price.unwrap(),
                started_at,
                ended_at,
                end_price,
                is_auction,
            );
        }
```

**POC**

https://gist.github.com/michaelbajor/5aadadf58ad52810903083b43245dab9

**PROPOSED SOLUTION**

It is highly recommended to change the *marketplace* contract's design so that it takes hold of NFTs when they are listed for sale. This can be achieved via *nft_transfer_call* process.

**REMEDIATION - ACKNOWLEDGED**

The team has acknowledged the issue stating that "As we don’t auction for the marketplace, we don’t need to worry about the loss funding by bid."

## GUV-9 Lack of Promise Callback Usage - Medium

It was observed that the *Promise* mechanism is used extensively throughout the codebase. However, the callbacks were not usually attached to them. The *Promise* mechanism is asynchronous, i.e. the cross-contract calls are executed in subsequent blocks and their failure does not revert the prior execution within the signle flow. It is up to the developers to manually verify if the *Promise* resolved successfuly and act acordingly depending on the result. As a general rule of thumb, most *Promises* should have an attached callback, unless otherwise specified by the business design. The most notable examples of missing callbacks are related to FT token transfers and deployments to newly created subaccounts. An exemplary code snippet that would beneift from callback mechanism is as follows:

vault/src/lib.rs:withdraw

```
if let Some(ft_contract) = &self.ft_contract {
    Promise::new(ft_contract.clone()).function_call(
        "ft_transfer".to_string(),
        json!({
            "receiver_id": owner.to_string(),
            "amount": amount_to_owner.to_string(),
        })
        .to_string()
        .into_bytes()
        .to_vec(),
        NearToken::from_yoctonear(1),
        Gas::from_tgas(20),
    );
```

vault/src/lib.rs:withdraw

```
            Promise::new(ft_contract.clone()).function_call(
                "ft_transfer".to_string(),
                json!({
                    "receiver_id": self.owner_contract.to_string(),
                    "amount": (amount_to_holders/2).to_string(),
                })
                .to_string()
                .into_bytes()
                .to_vec(),
                NearToken::from_yoctonear(1),
                Gas::from_tgas(20),
            );

        Promise::new(ft_contract.clone()).function_call(
            "ft_transfer".to_string(),
            json!({
                "receiver_id": treasury.clone().to_string(),
                "amount": (amount_to_holders/2).to_string(),
            })
            .to_string()
            .into_bytes()
            .to_vec(),
            NearToken::from_yoctonear(1),
            Gas::from_tgas(20),
        );
        Promise::new(ft_contract.clone()).function_call(
            "storage_withdraw".to_string(),
            json!({}).to_string().into_bytes().to_vec(),
            NearToken::from_yoctonear(1),
            Gas::from_tgas(20),
        );
    }
```

In the example above, it is crucial that each promise executes successfully. Furthermore, it is also important that the promise with *storage_withdraw* call is executed after the token transfers.

**PROPOSED SOLUTION**

It is recommended to attach a callback to every *Promise* that could fail and revert the state changes accordingly, if applicable. Usage of batch transactions is also highly encouraged to simplify the process.

**REMEDIATION - PARTIALLY FIXED**

Some of the Promises now utilize the callback mechanism, but not all. Namely:

- The NFT creation in the *launchpad* contract does not have an attached callback. This instance is remediated by making only*admin* use the functionality.
- The token transfers scheduled in NFT's *resolve_create* function are not checked if executed successfuly
- The Promise scheduled in the NFT's *burn* function with a call to a *withdraw* in the NFT is not checked if executed successfully

## GUV-10 Missing assert_one_yocto - Low

The *withdraw* function defined in the *nft* contract is marked as*payable*, however it does not verify the deposit. The *assert_one_yocto* function call is missing that would trigger a 2FA process when function is called manually. Similar case was observed for the *burn* function in the *nft* contract and in *set_config* function in *launchpad* contract.

### PROPOSED SOLUTION

It is recommended to add an *assert_one_yocto* call to the *withdraw* function to make sure that, when called directly, it was called using the Full Access key.

### REMEDIATION - FIXED

The *assert_one_yocto* function is called in each listed instance.

## GUV-11 Suboptimal Implementation Of Storage Staking Mechanism - Low

As per the design described by the development team, the *nft* contract is intended to deliberately not implement manual storage staking. The fees associated with holding the NFT are solely reliant on the original NFT creator. However, the *nft* contract does implement the *storage_deposit* function and keeps track of the storage deposits associated with various users, although those deposits are not checked anywhere else in the contract. Additionally, no storage withdrawal mechanism is implemented. If an uninformed user would use the *storage_deposit* functionality, every native token deposisted in that manner would be locked within the *nft* contract.

### PROPOSED SOLUTION

Since lack of storage staking is by-design, it is recommended to remove the *storage_deposit* function along with the *storage_deposits* map from the contract's state so that users wouldn't be able to lock their tokens within the uncomplete storage staking mechanism. An additionaly benefit is that the compiled binary size will be decreased.

### REMEDIATION - FIXED

The issue was resolved by removing the incomplete storage staking implementation

## GUV-12 Possible Precision Loss - Low

Due to the blockchain blockchain technology's very nature, every mathematical operation and data representation operates on integer numbers. However, operating only on those is inherently associated with a risk of losing precision. Although sometimes it is not possible to completely eradicate those issues, there are some measures that can be implemented to limit the impact of integer math.

The potential for precision loss was identified in:

- Division before multiplication. The *add_bid* functionality in the *marketplace* contract requires subsequent bids to be at least 10% higher then the previous bids. Dividing by *100* before multiplying by *10* might not result in a less accurate calculation then first multiplying and then dividing.

marketplace/src/lib.rs:add_bid

```
assert!(
  amount.0 >= current_bid.price.0 + (current_bid.price.0 / 100 * 10),
   "DS: Can't pay less than or equal to current bid price + 10% : {:?}",
   current_bid.price.0 + (current_bid.price.0 / 100 * 10)
);
```

- Assuming the number will always be divisible by *2*. The *withdraw* function in the *vault* contract transfers tokens, depending on the configuration either fungible or native, to various users defined in the contract, namely the contract's owner and the treasury. Both are transferred the amount equal to *amount_to_holders / 2*. If the *amount_to_holders* is not divisible by *2* there will some residual tokens in the *vault* contract's balance, which in case of FT tokens - will not be recoverable. Although the actual value of those residual tokens will likely be negligible, it will impact the accounting.

vault/src/lib.rs:withdraw

```
    if let Some(ft_contract) = &self.ft_contract {
            Promise::new(ft_contract.clone()).function_call(
              "ft_transfer".to_string(),
              json!({
                  "receiver_id": owner.to_string(),
                  "amount": amount_to_owner.to_string(),
              })
              .to_string()
              .into_bytes()
              .to_vec(),
              NearToken::from_yoctonear(1),
              Gas::from_tgas(20),
            );
            Promise::new(ft_contract.clone()).function_call(
              "ft_transfer".to_string(),
              json!({
                  "receiver_id": self.owner_contract.to_string(),
              "amount": (amount_to_holders/2).to_string(),
              })
              .to_string()
              .into_bytes()
              .to_vec(),
              NearToken::from_yoctonear(1),
              Gas::from_tgas(20),
            );
```

vault/src/lib.rs:withdraw

```
            Promise::new(ft_contract.clone()).function_call(
                "ft_transfer".to_string(),
                json!({
                    "receiver_id": treasury.clone().to_string(),
                 "amount": (amount_to_holders/2).to_string(),
                })
                .to_string()
                .into_bytes()
                .to_vec(),
                NearToken::from_yoctonear(1),
                Gas::from_tgas(20),
            );
            Promise::new(ft_contract.clone()).function_call(
                "storage_withdraw".to_string(),
                json!({}).to_string().into_bytes().to_vec(),
                NearToken::from_yoctonear(1),
                Gas::from_tgas(20),
            );
        }
        (...)
```

**PROPOSED SOLUTION**

It is recommended to make sure all multiplication is done before the division, whenever possible. Additionally, it is advised to consider usageo of *div_ceil* function to assure that user will always need to pay at least 10% more.

Additionally, in case of transferring the balances that require division, it is recommended at the last step to transfer the rest of the amount. Exemplary code that would calculate transfer amounts is as follows:

example

```
let first_transfer_amount = amount_to_holders / 2;
let second_transfer_amount = amount_to_holders - first_transfer_amount;
```

**REMEDIATION - PARTIALLY FIXED**

Some of the recommended improvements were implemented.

## GUV-13 Possible Token Lock In The Vault - Low

The *vault* contract utilizes *ft_on_transfer* function to receive the fungible tokens. However, the *vault* might be configured to be operating only on native NEAR tokens. The NEP141 standard specifies that the *ft_on_transfer* function should return the amount of not used tokens so that the token contract can return them to the sender. It was observed that even if *vault* is always returning *0* in the *ft_on_transfer* function stating that every transferred token was used within the protocol regardless if it was configured to work only with native tokens or not. Such a scenario results in locking the tokens within the *vault* contract.

**PROPOSED SOLUTION**

It is recommended to reject every token transfer if the *vault* is configured to operate on native tokens. Such rejection can be implemented by returning the exact amount of transferred tokens from the *ft_on_transfer* function so that the token contract will reimburse the full transferred amount to the sender.

**REMEDIATION - FIXED**

The issue was resolved.

## GUV-14 Hardcoded Storage Cost - Low

The *launchpad* contract uses a const value for storage byte cost. This value is set to an actual current value, however, the NEAR blockchain does not guarantee that this value won't change in the future. Hardcoding the value within the code would require a complete contract upgrade in order to change it. This can lead to miscalculating the required deposit, should this value ever change.

### PROPOSED SOLUTION

It is recommended to use *env::storage_byte_cost* function whenever this value is required instead of setting it a constant value.

### REMEDIATION - FIXED

The issue was resolved.

## GUV-15 Unverified Calculation Parameters - Informational

The *vault* contract defines a *withdraw* function which takes a *burn_fee* as one of the arguments. This function is only callable by the *nft* contract within the *burn* functions execution. Hence, the value of *burn_fee* will be equal to the one saved within the *nft* contract. However, the *burn_fee* is not verified to be within the safe range when it is set. It is worth noting that although not verification is in place, the only possible way to set the value is via an initialization function called by a deploying contract. Current version of the code hard-codes value for the *burn_fee* and it is within the safe range. However, should any contract upgrade take place and that value would change, accidentally or deliberately, it might result in an invalid protocol configuration.

Analogous situation happens for the *payment_split_percent* value, also set in the *nft* contract initialization function.

**PROPOSED SOLUTION**

It is recommended to always verify every value that is part of the protocol's configuration. Adding a separate setter function for those values should also be considered.

**REMEDIATION - FIXED**

The values for *payment_split_percent* and *burn_fee* are now parameters in the *launch* function of the *launchpad* contract. Additionally, they are verified that both add up to a value less than **100**.

## GUV-16 Miscellaneous Notes - Informational

This entry is a collection of general notes about the code and possible improvements that could be implemented. They are not inherently associated with security, but might impact the performance and gas efficiency of the contract.

- The *ft* contract contains the *new_default_meta* function used to initialize the contract with exemplary values, which likely will never be used in production. This function can savely be removed to decrease the compiled binary size.
- The *accept_bid* function defined in *marketplace* contract contains *println!* macro, which is not reflected wihtin the NEAR blockchain execution environement. That macro call can safely be deleted to reduce the compiled binary size.
- The *internal_cancel_bid* function defined in the *marketplace* contract contains an *assert!* statement that verifies *bids* vector's length to be positive. However, this function is called only in the context where that assertion has already been checked directly or indirectly making it redundant. It can be safely deleted from the *internal_cancel_bid* unless future development will require it.
- Several contracts in the code base contain a *require* statement that assures state does not exist within the functions marked with *init* macro. The *init* macro is responsible for doing the same assertion making the manual one redundant. The impacted functions are: *new* function in the *nft* contract and *new* function in the *ft* contract.
- The *assert* macro is used in multiple places within the codebase. Although it does work correctly, it also introduces extensive debug information about the line of code that caused a *panic*. This debug information is not needed within the NEAR blockchain's execution environment while also unnecessarily increasing the compiled binary size. Rust *assert* macros can be safely replaced with NEAR SDK's *require* macros which will also cause a *panic* on failed assertion, but without bloating the binary size.
- The *nft_mint* function is using an *if let Some(_) = self.mint_currency.clone()* statement. The *clone* operation is usually expensive and since the value contained in the *Some* variant is not used, it can be replaced with *if self.min_currency.is_some()*.
- The *nft* contract's *burn* function uses a so-called magic number, i.e. a specific constant value used during the protocol's execution, however without an explicit explanation. Existance of such values does not automatically introduce a security threat, however it is recommended to store every hard-coded value as a *const* variable to increase the readability and maintainability of the codebase.