Guvenkaya

PUBLIC REPORT
1ST JUNE 2024

SECURITY ASSESSMENT REPORT

NEAR Rust Smart Contract Security Assessment

Guvenkaya reviewed the Ample Protocol NFT series and treasury smart contracts on NEAR.

Prepared for
Ample Protocol

Client
Ample Protocol

Review period
27th February 2024 - 6th March 2024

Review team
Timur Güvenkaya - Principal

17	10	4	3	15
Findings raised	Critical or high	Medium	Low or info	Fixed in review

Contents

Every contents row and every underlined finding ID is clickable.

00	Engagement Dashboard	01
00	Review Team	02
01	Advisory and Client	03
02	Scope and Review Boundaries	04
03	Assessment Framework	06
04	Findings Summary	09
05	Detailed Findings	11
GUV-1	Anyone Can Update The NFT Holder In Treasury	11
GUV-2	Anyone Can Add Analytics	12
GUV-3	Anyone Can Add Content	13
GUV-4	DoS of Adding Analytics Functionality Due To Gas Limit	13
GUV-5	DoS of Adding Analytics Functionality Due To Log Limit	16
GUV-6	Storage Key Collisions Due To Missing Separator	16
GUV-7	Multiple Storage Key Collisions Due To Block Timestamp Usage	20
GUV-8	Users Can Transfer Ownership In Treasury Without Sending NFT	22
GUV-9	Users Can Create Series With Arbitrary Content Ids	23
GUV-10	Unpayable NFT Royalties Due To Underflow	23
GUV-11	Incorrect Cuts Calculation Due To Division Before Multiplication	24
GUV-12	Usage Of Floating Point Arithmetic In The Smart Contract	25
GUV-13	NFT Contract Is Not Built Using NEAR Contract Standards	25
GUV-14	Owner of NFT Can Set Arbitrary Royalty Payout Object	26
GUV-15	Privileged Functions Are Not Protected By 2FA	27
GUV-16	Suboptimal Assertion Usage	27
GUV-17	Contract Size Can Be Decreased	28

Engagement Dashboard

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, 1 low and 2 informational severity. All major issues are fixed by the Ample team.

ENGAGEMENT RECORD

CLIENT	Ample Protocol
SYSTEM	Ample NFT series and treasury smart contracts on NEAR
REVIEW PERIOD	27th February 2024 - 6th March 2024
DRAFT REPORT	6th March 2024
FINAL REPORT	1st June 2024
FILES IN SCOPE	19 source files
PRINCIPAL ADVISOR	Timur Güvenkaya

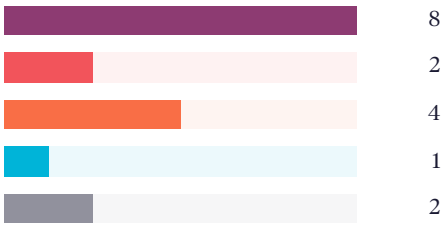
IMPACT

8	6	3
SEVERE	MODERATE	NEGLIGIBLE

LIKELIHOOD

11	3	3
LIKELY	POSSIBLE	RARE

FINDINGS BY SEVERITY



FINDING STATUS AT FINAL REPORT

17	15	2
RAISED	FIXED	ACKNOWLEDGED

REMEDIATION COMPLEXITY

12	3	2
SIMPLE	MODERATE	COMPLEX

TIMELINE

01 Review start
27th February 2024

02 Draft report
6th March 2024

03 Final report
1st June 2024

Review Team

The principal advisor named on this engagement. The profile is the public Guvenkaya biography for that reviewer.

Timur Güvenkaya

Principal advisor

Rust · NEAR & Substrate · Custody

Timur founded Guvenkaya after seeing teams reduce security to code review while their real risk spans architecture, infrastructure, operations, custody, and launch decisions. Before Guvenkaya, he established and led a security engineering practice for complex blockchain systems, specializing in Rust-based and non-EVM ecosystems including Substrate and NEAR. Earlier at Invicti, he helped build enterprise vulnerability-scanning and security detection engines used by Fortune 50 companies and public-sector organizations.

Advisory and Client

Guvenkaya

THE ADVISORY

Guvenkaya

Guvenkaya specializes in security for digital assets and financial infrastructure, from smart contracts and protocols to applications, infrastructure, custody and cryptography.

Our team, including the sitting Heads of Security at Gauntlet and Agora, brings experience from Kraken, OpenZeppelin, Binance, ZKsync, Nillion, Invicti and EY. Across the team: 200+ security reviews, $30B+ in volume secured, 13 published cryptography research papers, and credentials including OSCP, OSWE and CISM.

- Every reviewer has a public bio and is named before you sign. We match expertise to your system, with a principal leading scoping and checking findings.

AMPLE

THE CLIENT

Ample Protocol

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

Scope and Review Boundaries

REVIEW DEPTHS

Deep	Line-by-line review with explicit attack hypotheses, adversarial reasoning about state transitions, and reproduction attempts for each hypothesis that survived reading.
Supporting	The code was read for its effect on a reviewed flow. Behaviour outside that effect was not assessed.
Targeted	Entry points were exercised or triaged against the flow's security properties rather than read in full.

This is the code we read, and how closely. 19 paths are in scope. The original report counts 2,471 lines of code reviewed. Each path links to the file at the reviewed commit.

PATH	REVIEW DEPTH	SCOPE NOTE
series/src/approval.rs	Deep	NFT Series Smart Contract
series/src/enumeration.rs	Deep	NFT Series Smart Contract
series/src/events.rs	Deep	NFT Series Smart Contract
series/src/internal.rs	Deep	NFT Series Smart Contract
series/src/lib.rs	Deep	NFT Series Smart Contract
series/src/metadata.rs	Deep	NFT Series Smart Contract
series/src/nft_core.rs	Deep	NFT Series Smart Contract
series/src/owner.rs	Deep	NFT Series Smart Contract
series/src/royalty.rs	Deep	NFT Series Smart Contract
series/src/series.rs	Deep	NFT Series Smart Contract
series/src/treasury_ext.rs	Deep	NFT Series Smart Contract
src/accounts.rs	Deep	Treasury Smart Contract
src/analytics.rs	Deep	Treasury Smart Contract
src/collections.rs	Deep	Treasury Smart Contract
src/content.rs	Deep	Treasury Smart Contract
src/lib.rs	Deep	Treasury Smart Contract
src/royalties.rs	Deep	Treasury Smart Contract

PATH	REVIEW DEPTH	SCOPE NOTE
src/types.rs	Deep	Treasury Smart Contract
src/utils.rs	Deep	Treasury Smart Contract

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.

Assessment Framework

Guvenkaya rates findings by likelihood and impact so remediation can be prioritized by technical risk, business context, and the decision the engagement is meant to support.

RATING CHAIN APPLIED TO EVERY FINDING

01
Likelihood
Likely, possible or rare

02
Impact
Severe, moderate or negligible

03
Severity
Derived from the matrix below

04
Remediation complexity
Simple, moderate or difficult

05
Status
Not fixed, fixed, acknowledged or scheduled

Impact

IMPACT	WHAT IT MEANS HERE
Severe	Exploitation could result in critical loss or compromise, such as full system control, substantial financial loss, or severe reputational damage.
Moderate	Exploitation may lead to limited data loss, partial compromise, moderate financial impact, or noticeable degradation of services.
Negligible	Exploitation has minimal impact, such as minor data exposure without significant consequences or slight inconvenience without substantial disruption.

Likelihood

LIKELIHOOD	WHAT IT MEANS HERE
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 difficult 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.

ATTACK COMPLEXITY \ DISCOVERY EASE	OBVIOUS	CONCEALED	HIDDEN
COMPLEX	Possible	Rare	Rare
MODERATE	Likely	Possible	Rare
STRAIGHTFORWARD	Likely	Possible	Possible

Severity

SEVERITY	WHAT IT MEANS HERE
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.

LIKELIHOOD \ IMPACT	SEVERE	MODERATE	NEGLIGIBLE
LIKELY	Critical	High	Medium
POSSIBLE	High	Medium	Low
RARE	Medium	Low	Informational

Remediation complexity

This measures how difficult it is to remediate a finding once it has been identified.

REMEDIATION COMPLEXITY	WHAT IT MEANS HERE
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.

Finding status

This tracks the client's response to each finding and whether remediation has been completed, accepted, or scheduled.

STATUS	WHAT IT MEANS HERE
Not Fixed	Indicates that the vulnerability has been identified 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, such as patching, configuration changes, or architectural modifications, 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 fixed. It indicates that the risk posed by the vulnerability is known and has been consciously accepted.
Scheduled	This status indicates that the vulnerability has been acknowledged and a plan is in place to fix it in the future. It signifies that while remediation hasn't yet occurred, the issue has been prioritized and is part of the planned development roadmap.

Findings Summary

8	2	4	1	2
Critical	High	Medium	Low	Informational

Findings register

FINDING	IMPACT	LIKELIHOOD	SEVERITY	REMEDIATION COMPLEXITY	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	STATUS
GUV-14 Owner of NFT Can Set Arbitrary Royalty Payout Object	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

Detailed Findings

GUV-1 Anyone Can Update The NFT Holder In Treasury

Severity	Impact	Likelihood
Critical	Severe	Likely
Remediation complexity	**Status**	
Simple	Fixed	

DESCRIPTION

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 · RUST

```rust
#[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 · RUST

```rust
#[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);
}
}
```

RECOMMENDATION

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

REMEDIATION

The Ample Protocol team has fixed the issue.

GUV-2 Anyone Can Add Analytics

Severity	Impact	Likelihood
Critical	Severe	Likely
Remediation complexity	**Status**	
Simple	Fixed	

DESCRIPTION

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 · RUST

```
#[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()
    ...
```

RECOMMENDATION

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

REMEDIATION

The Ample Protocol team has fixed the issue.

GUV-3 Anyone Can Add Content

Severity	Impact	Likelihood
Critical	Severe	Likely
Remediation complexity	**Status**	
Simple	Fixed	

DESCRIPTION

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 · RUST

```
#[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(),
            ));
        ...
```

RECOMMENDATION

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

REMEDIATION

The Ample Protocol team has fixed the issue.

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

Severity	Impact	Likelihood
Critical	Severe	Likely
Remediation complexity	**Status**	
Simple	Fixed	

DESCRIPTION

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 · RUST

```rust
// 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 · RUST

```rust
#[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,
    };

    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()?;
```

POC:DOS_OF_ADDING_ANALYTICS (CONTINUED) · RUST

```
            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,
            };

                            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,
                            };
```

POC:DOS_OF_ADDING_ANALYTICS (CONTINUED) · RUST

```
                            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(())}
```

RECOMMENDATION

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

REMEDIATION

The Ample Protocol team has fixed the issue.

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

Severity	Impact	Likelihood
Critical	Severe	Likely
Remediation complexity	**Status**	
Simple	Fixed	

DESCRIPTION

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 · RUST

```rust
// 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);
}
```

RECOMMENDATION

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

REMEDIATION

The Ample Protocol team has fixed the issue.

GUV-6 Storage Key Collisions Due To Missing Separator

Severity	Impact	Likelihood
Critical	Severe	Likely
Remediation complexity	**Status**	
Moderate	Fixed	

DESCRIPTION

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:

RUST

```
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 · RUST

```
// 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 · RUST

```rust
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();

        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);
```

POC:SERIES_HASH_COLLISSION (CONTINUED) · RUST

```
        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,
        };

        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:#?}");
```

POC:SERIES_HASH_COLLISSION (CONTINUED) · RUST

```
        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:#?}");

        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(())}
```

RECOMMENDATION

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

REMEDIATION

The Ample Protocol team has fixed the issue.

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

Severity	Impact	Likelihood
Critical	Severe	Likely
Remediation complexity	**Status**	
Moderate	Fixed	

DESCRIPTION

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 · RUST

```
1  else {
2      let mut new_analytics = Vector::new(env::block_timestamp().try_to_vec().unwrap());
3      new_analytics.push(&new_analytic);
4
5      self.analytics_by_content_id
6          .entry(content_id.clone())
7          .or_insert(new_analytics);
8  }
9  ...
10 let mut team_members: UnorderedMap<AccountId, u8> =
11     UnorderedMap::new(env::block_timestamp().try_to_vec().unwrap());
12 let mut holders_by_token_id: UnorderedMap<String, AccountId> =
13     UnorderedMap::new(env::block_timestamp().try_to_vec().unwrap());
14     ...
```

TREASURY:CLAIM_ROYALTIES:TREASURY/SRC/ROYALTIES.RS · RUST

```
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 · RUST

```
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 · RUST

```
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()),
    );
    ...
```

RECOMMENDATION

Consider creating unique keys for storage using unique identifiers.

REMEDIATION

The Ample Protocol team has fixed the issue.

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

Severity	Impact	Likelihood
Critical	Severe	Likely
Remediation complexity	**Status**	
Simple	Fixed	

DESCRIPTION

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 · RUST

```
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);
    ...
```

RECOMMENDATION

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

The Ample Protocol team has fixed the issue.

GUV-9 Users Can Create Series With Arbitrary Content Ids

Severity	Impact	Likelihood
High	Moderate	Likely
Remediation complexity	**Status**	
Simple	Fixed	

DESCRIPTION

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 · RUST

```
1  pub fn update_holder(
2      &mut self,
3      content_id: ContentId,
4      token_id: TokenId,
5      new_holder: AccountId,
6  ) {
7      // 1. Retrieve the collection using collection_by_content_id
8      let collection = self.collection_by_content_id.get_mut(&content_id).unwrap();
```

RECOMMENDATION

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

REMEDIATION

The Ample Protocol team has fixed the issue.

GUV-10 Unpayable NFT Royalties Due To Underflow

Severity	Impact	Likelihood
High	Moderate	Likely
Remediation complexity	**Status**	
Simple	Fixed	

DESCRIPTION

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 · RUST

```
1  // payout to previous owner who gets 100% - total perpetual royalties
2  payout_object.payout.insert(
3      owner_id,
4      royalty_to_payout(10000 - total_perpetual, balance_u128),
5  );
```

RECOMMENDATION

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

REMEDIATION

The Ample Protocol team has fixed the issue.

GUV-11 Incorrect Cuts Calculation Due To Division Before Multiplication

Severity	Impact	Likelihood
Medium	Negligible	Likely
Remediation complexity	**Status**	
Simple	Fixed	

DESCRIPTION

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 · RUST

```
1  pub(crate) fn calculate_cuts(percentage: f64, streams: f64) -> f64 {
2      let result: f64 = (percentage as f64 / 100.0) * streams;
3      result
4  }
```

RECOMMENDATION

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

REMEDIATION

The Ample Protocol team has fixed the issue.

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

Severity	Impact	Likelihood
Medium	Moderate	Possible
Remediation complexity	**Status**	
Moderate	Fixed	

DESCRIPTION

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 · RUST

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

RECOMMENDATION

Consider rewriting all the code with fixed point arithmetic.

REMEDIATION

The Ample Protocol team has fixed the issue.

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

Severity	Impact	Likelihood
Medium	Moderate	Possible
Remediation complexity	**Status**	
Complex	Acknowledged	

DESCRIPTION

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.

RECOMMENDATION

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

REMEDIATION

The Ample Protocol team has acknowledged the issue.

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

Severity	Impact	Likelihood
Medium	Moderate	Possible
Remediation complexity	**Status**	
Complex	Acknowledged	

DESCRIPTION

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 · RUST

```
1  fn nft_transfer_payout(
2      &mut self,
3      receiver_id: AccountId,
4      token_id: TokenId,
5      approval_id: u64,
6      memo: Option<String>,
7      balance: U128,
8      max_len_payout: u32,
9  ) -> Payout {
10 ...
11 if royalty_option.is_none() {
12     let mut payout = HashMap::new();
13     payout.insert(owner_id, balance);
14     return Payout { payout };
15 }
```

RECOMMENDATION

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

The Ample Protocol team has acknowledged the issue.

GUV-15 Privileged Functions Are Not Protected By 2FA

Severity	Impact	Likelihood
Low	Moderate	Rare
Remediation complexity	**Status**	
Simple	Fixed	

DESCRIPTION

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.

RECOMMENDATION

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

REMEDIATION

The Ample Protocol team has fixed the issue.

GUV-16 Suboptimal Assertion Usage

Severity	Impact	Likelihood
Informational	Negligible	Rare
Remediation complexity	**Status**	
Simple	Fixed	

DESCRIPTION

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 · RUST

```
1  //make sure we're not paying out to too many people (GAS limits this)
2  assert!(
3      royalty.len() as u32 <= max_len_payout,
4      "Market can&not payout to that many receivers"
5  );
```

RECOMMENDATION

Consider using require instead of assert.

REMEDIATION

The Ample Protocol team has acknowledged the issue.

GUV-17 Contract Size Can Be Decreased

Severity	Impact	Likelihood
Informational	Negligible	Rare
Remediation complexity	**Status**	
Simple	Fixed	

DESCRIPTION

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 · RUST

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

RECOMMENDATION

Consider removing rlib to decrease the contract size.

REMEDIATION

The Ample Protocol team has fixed the issue.