KUDELSKI SECURITY

Security Assessment for Audius Protocol

Findings and Recommendations Report Presented to:

Audius

October 27, 2021

Version: 1.1.0

Presented by:

Kudelski Security, Inc.
5090 North 40th Street, Suite 450
Phoenix, Arizona 85018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

Overview

Audius engaged Kudelski Security to perform a Security Assessment for Audius Protocol.

The assessment was conducted remotely by the Kudelski Security Team. Testing took place on September 27 - October 26, 2021, and focused on the following objectives:

- Provide the customer with an assessment of their overall security posture and any risks that were discovered within the environment during the engagement.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the result of our tests.

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the Kudelski Security Teams took to identify and validate each issue, as well as any applicable recommendations for remediation.

Key Findings

The following are the major themes and issues identified during the testing period. These, along with other items, within the findings section, should be prioritized for remediation to reduce to the risk they pose.

- KS-AUDPROT-01 – RewardsManager - Missing signature check
- KS-AUDPROT-02 – ClaimableTokens - Missing signature check
- KS-AUDPROT-03 – ClaimableTokens - Token transfer can be replayed
- KS-AUDPROT-04 – ClaimableTokens - Amount transferred is not signed
- KS-AUDPROT-05 – ClaimableTokens - Immutability of min_votes
- KS-AUDPROT-06 – EthRewardsManager.sol - Token transfer is always approved and can be invoked without restrictions.
- KS-AUDPROT-07 – WormholeClient.sol - Redundant signature check
- KS-AUDPROT-08 – find_program_address can cause transactions to fail
- KS-AUDPROT-09 – ClaimableTokens - Derived address created in Claimable Tokens could be owned
- RewardsManager - Token transfer from attestations can be stolen

During the test, the following positive observations were noted regarding the scope of the engagement:

- The team was very supportive and open to discuss the design choices made

Based on the account relationship graphs or reference graphs and the formal verification we can conclude that the reviewed code implements the documented functionality.

Scope and Rules of Engagement

Kudelski performed a Security Assessment for Audius Protocol. The following table documents the targets in scope for the engagement. No additional systems or resources were in scope for this assessment.

The source code was supplied through a public repository at https://github.com/AudiusProject/audius-protocol with the commit hash 5479793f37eea25fbba0f8e5a4d5e81b1b82bbfb, and with commit hash 2c93f29596a1d6cc8ca4e76ef1f0d2e57f0e09e6 after remediations.

The specific files in scope are listed in Figure 1 below.

Files included in the code review

```
solana-programs
├── claimable-tokens/
│   ├── cli/
│   │   ├── src/
│   │   │   └── main.rs
│   └── program/
│       ├── src/
│       │   ├── utils/
│       │   │   ├── mod.rs
│       │   │   └── program.rs
│       │   ├── entrypoint.rs
│       │   ├── error.rs
│       │   ├── instruction.rs
│       │   ├── lib.rs
│       │   ├── processor.rs
│       │   └── state.rs
├── reward-manager/
│   ├── cli/
│   │   ├── src/
│   │   │   ├── main.rs
│   │   │   └── utils.rs
│   │   └── Cargo.toml
│   ├── program/
│   │   ├── src/
│   │   │   ├── state/
│   │   │   │   ├── mod.rs
│   │   │   │   ├── reward_manager.rs
│   │   │   │   ├── sender_account.rs
│   │   │   │   └── verified_messages.rs
│   │   │   ├── utils/
│   │   │   │   ├── mod.rs
│   │   │   │   └── signs.rs
│   │   │   ├── entrypoint.rs
│   │   │   ├── error.rs
│   │   │   ├── instruction.rs
│   │   │   ├── lib.rs
│   │   │   └── processor.rs
│   │   ├── Cargo.toml
│   │   └── Xargo.toml
│   └── Cargo.toml
└── eth-contracts/
    └── contracts/
        ├── DelegateManagerV2.sol
        ├── EthRewardsManager.sol
        └── WormholeClient.sol
```

Table 1: Scope

TECHNICAL ANALYSIS & FINDINGS

During the Security Assessment for Audius Protocol, we discovered:

- 4 findings with HIGH severity rating.
- 1 finding with MEDIUM severity rating.
- 2 findings with LOW severity rating.
- 3 findings with INFORMATIONAL severity rating.

The following chart displays the findings by severity.

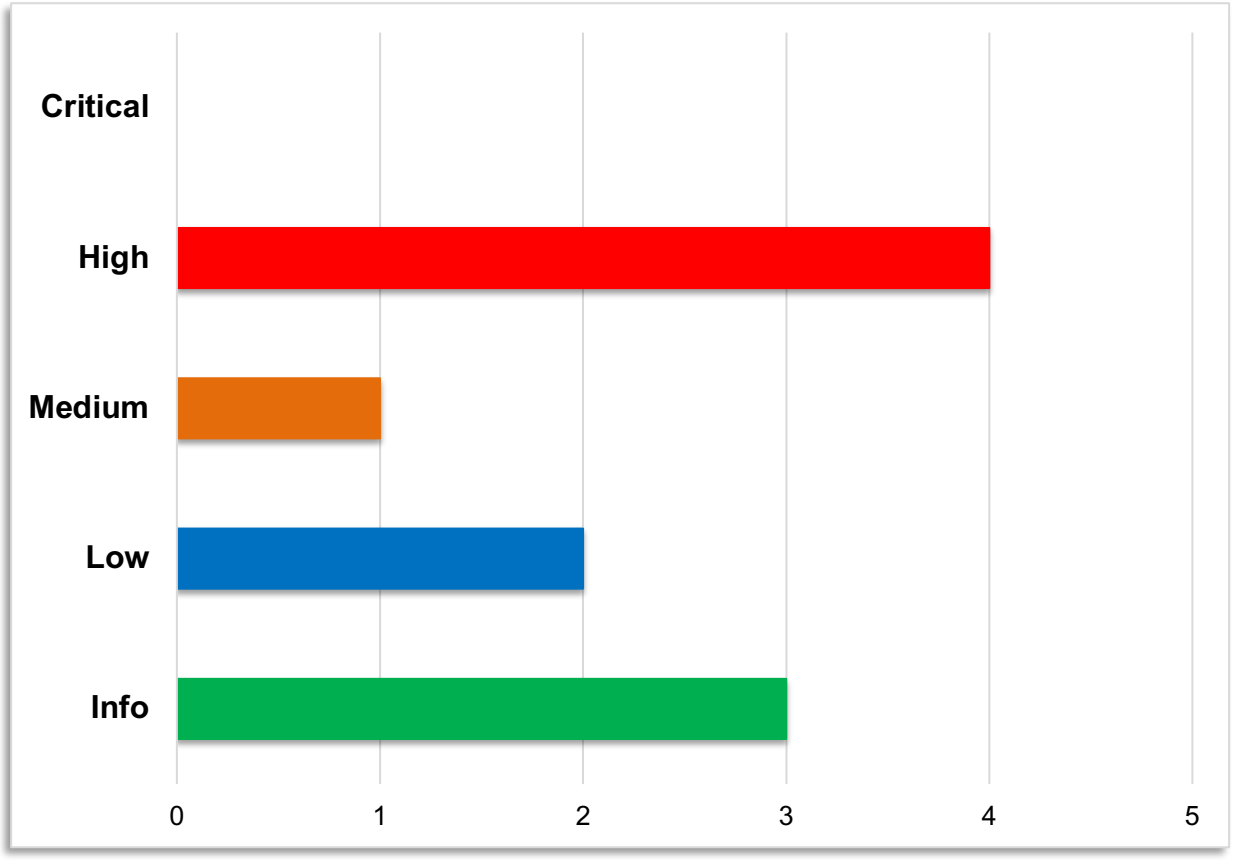

Figure 1: Findings by Severity

Findings

The *Findings* section provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

#	Severity	Description
KS-AUDPROT-01	High	RewardsManager - Missing signature checks
KS-AUDPROT-02	High	ClaimableTokens - Missing signature check
KS-AUDPROT-03	High	ClaimableTokens - Token transfer can be replayed
KS-AUDPROT-04	Medium	ClaimableTokens - Amount transferred is not signed
KS-AUDPROT-05	Low	RewardsManager - Immutability of min_votes
KS-AUDPROT-06	Low	EthRewardsManager.sol - Token transfer is always approved and can be invoked without restrictions.
KS-AUDPROT-07	Informational	WormholeClient.sol - Redundant signature check
KS-AUDPROT-08	Informational	find_program_address can cause transactions to fail
KS-AUDPROT-09	Informational	ClaimableTokens - Derived address created in Claimable Tokens could be owned
KS-AUDPROT-10	High	RewardsManager - Token transfer from attestations can be stolen

Table 2: Findings Overview

Technical analysis

Based on the source code the validity of the code was verified as well as confirming that the intended functionality was implemented correctly and to the extent that the state of the repository allowed. A number of further investigations were made which concluded that they did not pose a risk to the application. They were:

- No potential authorization issues were observed
- No internal unintentional unsafe references
- No large memory allocations
- No unintended loss of precision

- No unjustified drop implementations
- No unjustified foreign function interface implementations

Conclusion

Based on formal verification we conclude that the code implements the documented functionality to the extent of the code reviewed.

Technical Findings

General Observations

Overview

Audius is a digital streaming service that connects music artists and fans, providing rewards in the form of an ERC-20 token $AUDIO. It does so by deploying and operating smart contracts and programs to the Ethereum and Solana main-net blockchains respectively.

The contents of these decentralized applications were the subject of this review. A total of 10 instructions across two Solana programs were audited, together with two Ethereum smart contracts and partial changes to a previously audited contract. These form Audius' Rewards System and DelegateManagerV2.

In general, the code was well written, documented, and tested. Questions were also promptly answered by the development team.

Ethereum Contracts

Ethereum contracts were generally more concrete, utilizing the blockchain's checks and constrains to secure related assets. Without having reviewed the entirety of the contract, changes to the DelegateManagerV2 seem to address the bug related to a missed round of rewards, by correctly updating `validBounds` after an un-delegation. Additional constrains introduced for the MinDelegation update are securely controlled. Comments on the other two contracts are listed as Informational findings below.

Solana Programs

Solana programs are well structured and documented. Instructions are unpacked early, while logic is clearly separated in different functions. Structure cloning could sometimes be avoided, e.g. when using immutable AccountInfo references. Another minor observation is that accounts are mostly created on-chain, calling `find_program_address` whose complications are described in detail as an informational finding below.

The major oversight however, leading to the high impact issues discussed below, is the unchecked integrity of account data passed as inputs to the instructions. This oversight is very visible and impactful during verification of Ethereum signatures. The checks are only partially implemented allowing attackers to bypass them using fabricated accounts, leading to unconstrained transfer of tokens, from program owned accounts. Additional integrity checks are missing and are discussed in the issues below.

RewardsManager - Missing signature checks

Finding ID: KS-AUDPROT-01
Severity: **High**
Status: **Remediated**

Description

Ethereum signatures are used extensively in the RewardsManager program for creating and deleting sender accounts as well as attesting to token transfers.

Signature verification is done by loading an instruction from data passed via the input account `instruction`. This account's data are de-serialized into an Instruction structure whose program id is checked to be `secp256k1_program::id()` – Notice that the ownership of the `instruction` account is not verified. The loaded instruction's data are then checked against other inputs:

- The signer is matched against senders' Ethereum addresses.
- The data being signed over are matched to an account creation/deletion or to an attestation.

Potential attackers could fabricate their own `instruction` account data - using known senders' Ethereum addresses, messages of their choosing, and arbitrary signature data.

Attackers could then invoke the program's instructions with the fabricated account and inputs matching their choices. Since the signature is not checked, partial verification will succeed, allowing the program to proceed to the creation/deletion of accounts, or the transfer of tokens from a program owned account to the attacker's selected account.

Proof of Issue

File name: solana-programs/reward-manager/program/src/utils/signs.rs
Line number: 165

```
pub fn validate_secp_add_delete_sender(
    program_id: &Pubkey,
    reward_manager: &Pubkey,
    instruction_info: &AccountInfo,
    expected_signers: Vec<&AccountInfo>,
    extraction_depth: usize,
    new_sender: EthereumAddress,
    message_prefix: &str,
) -> ProgramResult {
    let index =
sysvar::instructions::load_current_index(&instruction_info.data.borrow());
    // Instruction can't be first in transaction
    // because must follow after `new_secp256k1_instruction`
    if index == 0 {
        return Err(AudiusProgramError::Secp256InstructionMissing.into());
    }

    // Load previous instructions
    let secp_instructions = get_secp_instructions(index, extraction_depth,
instruction_info)?;
```

```
    // Get the eth addresses associated with our expected_signers
    let (senders_eth_addresses, _) =
        get_and_verify_signer_metadata(program_id, reward_manager,
expected_signers)?;

    let mut checkmap = vec_into_checkmap(&senders_eth_addresses);
    let expected_message = [
        message_prefix.as_ref(),
        reward_manager.as_ref(),
        new_sender.as_ref(),
    ]
    .concat();

    // For each secp instruction, assert that the signer was expected and not
duplicated
    // and that the message is formatted correctly.
    for secp_instruction in secp_instructions {
        let eth_signer =
get_signer_from_secp_instruction(secp_instruction.data.clone());
        check_signer(&mut checkmap, &eth_signer)?;
        check_message_from_secp_instruction(secp_instruction.data,
expected_message.as_ref())?;
    }

    Ok(())
}

/// Checks secp instruction for submit_attestation:
/// ensures the message is signed by `expected_signer`, and
/// returns the message.
pub fn validate_secp_submit_attestation(
    instruction_info: &AccountInfo,
    expected_signer: &EthereumAddress,
) -> Result<VoteMessage, ProgramError> {
    let index =
sysvar::instructions::load_current_index(&instruction_info.data.borrow());

    // Instruction can't be first in transaction
    // because must follow after `new_secp256k1_instruction`
    if index == 0 {
        return Err(AudiusProgramError::Secp256InstructionMissing.into());
    }

    // Load previous instruction
    let secp_instruction = sysvar::instructions::load_instruction_at(
        (index - 1) as usize,
        &instruction_info.data.borrow(),
    )
    .map_err(to_audius_program_error)?;

    // Check that instruction is `new_secp256k1_instruction`
    if secp_instruction.program_id != secp256k1_program::id() {
        return Err(AudiusProgramError::Secp256InstructionMissing.into());
```

```
    }

    let eth_signer =
get_signer_from_secp_instruction(secp_instruction.data.clone());
    if eth_signer != *expected_signer {
        return Err(AudiusProgramError::WrongSigner.into());
    }

    get_vote_message_from_secp_instruction(secp_instruction.data)
}
```

To see the problem we can first look at `sysvar::instructions::load_current_index` to notice that this simply returns the last two bytes of its input as `u16`.

The following invocation to `sysvar::instructions::load_instruction_at` simply calls `message::Message::deserialize_instruction`.

The latter simply de-serializes the data at a given index of the memory passed, and returns a constructed Instruction struct. The only place it fails is if the memory is not structured correctly, e.g. the length of the data is invalid.

During `validate_secp_add_delete_sender`, the data of the de-serialized instructions are only partially checked:

- The signer is checked against the expected Ethereum address and
- The data being signed over are checked against instruction inputs.

Also note that the signed data of are not checked in `validate_secp_submit_attestation`.

An attacker could fabricate an `instruction` account, serializing data containing senders' Ethereum address and messages of their choosing. Invoking the instruction with this account and respective inputs, would allow the attacker to bypass the expected signature check.

Severity and Impact summary

Attackers can abuse missing signature checks to create/delete account and fake attestations leading to unwanted token transfers from program owned accounts.

Remediation

The recommendation in the report's initial draft, was more complicated than the one currently implemented. To avoid the attack, the program now simply checks for ownership of `instruction_info.key` by the `sysvar::instructions`. This ensures the integrity of the check of the instruction's `program_id` against `secp256k1_program::id`. In addition, the offsets of the instruction's data, which can potentially hold many signatures, are checked to verify that only one signature, with expected offsets, is held in the account's data.

References

- `sysvar::instructions::load_current_index`
- `sysvar::instructions::load_instruction_at`

- `message::Message::deserialize_instruction`
- `solana_sdk::secp256k1_instruction::verify_eth_addresses`

ClaimableTokens - Missing signature check

Finding ID: KS-AUDPROT-02
Severity: **High**
Status: **Remediated**

Description

As we understand, instruction `TokenTransfer` of the ClaimableTokens program, should verify a signature produced by and linked to an Ethereum address from which a program owned account is derived.

Verification uses an `instruction` account, passed as input to the instruction. This account's data is de-serialized into an Instruction whose program id matches `secp256k1_program::id()` – Notice that the ownership of the `instruction` account is not verified. The loaded instruction's data are then checked against inputs of the `TokenTransfer`:

- The signer is matched against the Ethereum address used for the program derived address.
- The data being signed over are matched to the token transfer's destination account.

A potential attacker could fabricate their own `instruction` account data using

- a user's Ethereum address (which derives an account owned by the program),
- a destination account of their choosing
- and arbitrary signature data serialized into a valid `secp256k1_program` Instruction.

They could then invoke the instruction with the fabricated account and inputs matching their choices.

Since the signature is not checked, partial verification will succeed, allowing the program to proceed to the transfer of tokens from an owned account to the attacker's selected account.

Proof of Issue

File name: solana-programs/claimable-tokens/program/src/processor.rs
Line number: 235

```
    /// Checks that the user signed message with his ethereum private key
    fn check_ethereum_sign(
        instruction_info: &AccountInfo,
        expected_signer: &EthereumAddress,
        expected_message: &[u8],
    ) -> ProgramResult {
        let index =
sysvar::instructions::load_current_index(&instruction_info.data.borrow());

        // instruction can't be first in transaction
        // because must follow after `new_secp256k1_instruction`
        if index == 0 {
            return
Err(ClaimableProgramError::Secp256InstructionLosing.into());
        }
```

```
        // load previous instruction
        let instruction = sysvar::instructions::load_instruction_at(
            (index - 1) as usize,
            &instruction_info.data.borrow(),
        )
        .map_err(to_claimable_tokens_error)?;

        // is that instruction is `new_secp256k1_instruction`
        if instruction.program_id != secp256k1_program::id() {
            return
Err(ClaimableProgramError::Secp256InstructionLosing.into());
        }

        Self::validate_eth_signature(expected_signer, expected_message,
instruction.data)
    }

    /// Checks that message inside instruction was signed by expected signer
    /// and message matches the expected value
    fn validate_eth_signature(
        expected_signer: &EthereumAddress,
        expected_message: &[u8],
        secp_instruction_data: Vec<u8>,
    ) -> Result<(), ProgramError> {
        let eth_address_offset = 12;
        let instruction_signer = secp_instruction_data
            [eth_address_offset..eth_address_offset +
size_of::<EthereumAddress>()]
            .to_vec();
        if instruction_signer != expected_signer {
            return
Err(ClaimableProgramError::SignatureVerificationFailed.into());
        }

        //NOTE: meta (12) + address (20) + signature (65) = 97
        let message_data_offset = 97;
        let instruction_message =
secp_instruction_data[message_data_offset..].to_vec();
        if instruction_message != *expected_message {
            return
Err(ClaimableProgramError::SignatureVerificationFailed.into());
        }

        Ok(())
    }
```

To see the problem, we can first look at `sysvar::instructions::load_current_index` to notice that this simply returns the last two bytes of `instruction_info` data as `u16`. The following invocation to `sysvar::instructions::load_instruction_at` simply calls `message::Message::deserialize_instruction`.

The latter simply de-serializes the data at a given index of the memory passed, and returns a constructed Instruction struct. The only place this fails is if the memory is not structured correctly, e.g. the length of the data is invalid.

During `validate_eth_signature`, the data of the de-serialized instruction are only partially checked:

- The signer is checked against the expected Ethereum address and
- The data being signed over are checked against instruction inputs.

An attacker could fabricate an `instruction` account, serializing data containing an existing user's Ethereum address and a destination account of their choosing. Invoking the instruction with this account and respective inputs, would allow the attacker to bypass the expected signature check.

Severity and Impact summary

Attackers can abuse missing signature check to transfer tokens from program owned accounts to arbitrary destinations.

Remediation

The recommendation in the report's initial draft, was more complicated than the one currently implemented. To avoid the attack, the program now simply checks for ownership of `instruction_info.key` by the `sysvar::instructions`. This ensures the integrity of the check of the instruction's `program_id` against `secp256k1_program::id`. In addition, the offsets of the instruction's data, which can potentially hold many signatures, are checked to verify that only one signature, with expected offsets, is held in the account's data.

References

- `sysvar::instructions::load_current_index`
- `sysvar::instructions::load_instruction_at`
- `message::Message::deserialize_instruction`
- `solana_sdk::secp256k1_instruction::verify_eth_addresses`

ClaimableTokens - Token transfer can be replayed

Finding ID: KS-AUDPROT-03
Severity: **High**
Status: **Remediated**

Description

Instruction invocations are part of the blockchain and are publicly available. An attacker could copy a successful invocation of `Transfer` and replay it. Since no record of transactions is kept, the replayed instruction would succeed.

A potential attacker could for example be a user to whom tokens from this program were transferred once.

Proof of issue

File name: solana-programs/claimable-tokens/program/src/processor.rs
Line number: 76

```
        Self::check_ethereum_sign(
            instruction_info,
            &eth_address,
            &destination_account_info.key.to_bytes(),
        )?;
```

Valid inputs to `check_ethereum_sign` will always be valid since it does not change any account's state.

This allows for token successful token transfers to be replayed.

Severity and Impact summary

A user that has received tokens using the `Transfer` instruction, could replay the instruction invocation, receiving tokens without new signatures.

Remediation

The recommendation provided in the initial draft report was not optimal. After examining the provided solution, the problem can be considered fixed. To avoid the attack, an account is created for each signer, with its address derived using the `program_id` and the signer's address. The account holds a nonce which is incremented with every successful signature. The current value of the nonce is also checked against the signed data. This safeguards from the attack described above. A small nuance still incurs, since a valid address for extremely few signers might not be derived from `find_program_address`. This chance was estimated at 1/(2^30) and was accepted as a risk with to small of a chance to occur.

ClaimableTokens - Amount transferred is not signed

Finding ID: KS-AUDPROT-04
Severity: **Medium**
Status: **Remediated**

Description

The amount of tokens transferred during `Transfer` is not included in the signed data. This would allow an attacker intercepting a valid invocation to alter the amount of tokens transferred. Since the amount is not included and checked in the signature, the altered transfer would successfully take place.

Proof of issue

File name: solana-programs/claimable-tokens/program/src/processor.rs
Line number: 72

```
        Self::check_ethereum_sign(
            instruction_info,
            &eth_address,
            &destination_account_info.key.to_bytes(),
        )?;
```

The amount of tokens to be transferred are not provided to `check_ethereum_sign`. An attacker could reuse the data from a previous `Transfer` instruction using the same `instruction_info`, `destination_account_info` and altering the amount to their choosing.

Since it is omitted in the signature, the transfer with the altered amount would be allowed.

Severity and Impact summary

Missing amount from signed data would allow a man in the middle to alter the amount transferred, given a valid signature for a different amount.

Recommendation

This issue is somewhat similar to the replay attack above. Hence, ownership check and nonce would solve most of the issue.

Furthermore, it is recommended to require the associated Ethereum address to sign the amount to be transferred with the destination. The amount should be passed to `check_ethereum_sign` which should verify the signature of both the destination and amount.

Remediation

The issue was solved by reading the amount from the signed data.

RewardsManager - Immutability of min_votes

Finding ID: KS-AUDPROT-05
Severity: **Low**
Status: **Open**

Description

Reward managers are initialized with a fixed number of `min_votes` required. A user or group of users obtaining `min_votes` once, could create infinitely many senders. These accounts could be used for providing attestations as well as creating and deleting other sender accounts. It would be useful in that case to have a controlled way of modifying `min_votes`

Proof of issue

File name: solana-programs/reward-manager/program/src/processor.rs

The `min_votes` value of the reward manager account is never modified after initialization.

Severity and Impact summary

A user or group of users obtaining `min_votes` once, could create infinitely many senders “unlocking” all public instructions.

Recommendation

Provide an additional instruction, similar to `process_change_manager_account` enabling the modification of a reward manager’s `min_votes`.

EthRewardsManager.sol - Token transfer is always approved and can be invoked without restrictions.

Finding ID: KS-AUDPROT-06
Severity: **Low**
Status: **Open**

Description

Function `transferToSolana` of `EthRewardsManager` is publicly accessible. It can be invoked by all users and contracts without any checks. The token transfer is always approved.

Proof of issue

File name: eth-contracts/contracts/EthRewardsManager.sol
Line number: 104

```
    function transferToSolana(uint256 arbiterFee, uint32 _nonce) external {
        _requireIsInitialized();

        uint256 balance = audiusToken.balanceOf(address(this));
        audiusToken.approve(address(wormhole), balance);

        wormhole.transferTokens(
            address(audiusToken),
            balance,
            1,
            recipient,
            arbiterFee,
            _nonce
        );
    }
```

The only check performed during the function is whether the contract is initialized. Nor the `tx.origin`, neither `msg.sender` are checked, thus anybody could invoke this function. Given sufficient balances, the function will always succeed.

Fees and nonce replay attacks should also be investigated, using an instance of the Wormhole relevant to that used in production.

Severity and Impact summary

An arbitrary user could initiate the transfer of Audius tokens to Solana blockchain at any time.

Recommendation

Restrict access to `transferToSolana` to `ClaimsManager`, `governanceAddress` and/or other privileged actors. This could for example be achieved, similarly to other functions of the contract, by using:

```
require(msg.sender == governanceAddress, ERROR_ONLY_GOVERNANCE);
```

WormholeClient.sol - Redundant signature check

Finding ID: KS-AUDPROT-07
Severity: **Informational**
Status: **Open**

Description

In function `transferTokens` the signature of the sender is securely checked, correctly including nonces. Even so, this can be considered redundant, since an approval of token transfer to the WormholeClient should precede the function invocation and would have to be signed for.

This can be kept as an additional check, incurring however, additional gas costs.

Proof of Issue

File name: eth-contracts/contracts/WormholeClient.sol
Line number: 91

```
    function transferTokens(
        address from,
        uint256 amount,
        uint16 recipientChain,
        bytes32 recipient,
        uint256 arbiterFee,
        uint deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public {
        uint32 nonce = nonces[from]++;

        // solium-disable security/no-block-members
        require(deadline >= block.timestamp, "WormholeClient: Deadline has
expired");
        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                DOMAIN_SEPARATOR,
                keccak256(
                    abi.encode(
                        TRANSFER_TOKENS_TYPEHASH,
                        from,
                        amount,
                        recipientChain,
                        recipient,
                        arbiterFee,
                        nonce,
                        deadline
                    )
                )
            )
        );
```

```
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(
            recoveredAddress != address(0) && recoveredAddress == from,
            "WormholeClient: Invalid signature"
        );

        transferToken.safeTransferFrom(from, address(this), amount);
        transferToken.approve(address(wormhole), amount);

        wormhole.transferTokens(
            address(transferToken),
            amount,
            recipientChain,
            recipient,
        arbiterFee,
            nonce
        );
    }
```

The call to `transferToken.safeTransferFrom(from, address(this), amount);` would fail if the source address has not approved a transfer to the WormholeClient. Such an approval would require `from`'s signature. Thus the manual signature check can be considered redundant.

Severity and Impact summary

Redundant signature check incurs additional gas costs.

Recommendation

If not needed, remove the signature check, allowing the transaction to fail during `transferToken.safeTransferFrom`.

find_program_address can cause transactions to fail

Finding ID: KS-AUDPROT-08
Severity: **Informational**
Status: **Open**

Description

The Solana library function `find_program_address` is used extensively in the program. The `find_program_address` function has a very high compute cost as can create up to 256 calls to `create_program_address` trying to "brute force" a valid bump seed that results in a valid program address that does not lie on the ED25519 elliptic curve.

> *The processes of finding a valid program address is by trial and error, and even though it is deterministic given a set of inputs it can take a variable amount of time to succeed across different inputs. This means that when called from an on-chain program it may incur a variable amount of the program's compute budget. Programs that are meant to be very performant may not want to use this function because it could take a considerable amount of time. Also, programs that area already at risk of exceeding their compute budget should also call this with care since there is a chance that the program's budget may be occasionally exceeded.*
>
> - docs.rs - solana_program::pubkey::find_program_address

The downside of using `create_program_address` is that there is about a 50 percent chance that a bump seed will generate a valid program address. This also means that if the bump seed is passed as input from the user, another bump seed can be injected. Depending on the specific instruction this can open for unauthorized access.

> *Program addresses are deterministically derived from a collection of seeds and a program id using a 256-bit pre-image resistant hash function. Program address must not lie on the ed25519 curve to ensure there is no associated private key. During generation an error will be returned if the address is found to lie on the curve. There is about a 50/50 chance of this happening for a given collection of seeds and program id. If this occurs a different set of seeds or a seed bump (additional 8 bit seed) can be used to find a valid program address off the curve.*
>
> - Solana Documentation - Calling Between Programs: Hash-based generated program addresses

To minimize compute cost without compromising security, the bump seed must be stored as part of the initialization instruction. To verify program addresses, `create_program_address` can then be used in combination with the stored bump seed.

Proof of Issue

File name: solana-programs/claimable-tokens/program/src/utils/program.rs
Line number: 50

```
pub fn find_base_address(mint: &Pubkey, program_id: &Pubkey) -> (Pubkey, u8)
{
    Pubkey::find_program_address(&[&mint.to_bytes()[..32]], program_id)
}
```

File name: solana-programs/reward-manager/program/src/utils/mod.rs
Line number: 120

```
pub fn find_program_address(program_id: &Pubkey, pubkey: &Pubkey) -> (Pubkey,
u8) {
    Pubkey::find_program_address(&[&pubkey.to_bytes()[..32]], program_id)
}
```

The functions above are used in all instructions audited.

Severity and Impact Summary

The compute cost of the the `find_program_address` may exceed the allowed budget causing transactions to fail.

Recommendation

To reduce the compute cost, the use of `find_program_address` should be minimized. If possible, the user should call `find_program_address` off-chain and pass the bump seed to the initialization instruction which stores it in a program account. Otherwise, the initialization instruction could call `find_program_address` as the only instruction.

Other instructions should then use the stored bump seed to verify derived program addresses passed as account input by calling `create_program_address` at a cheaper compute cost.

References

- Solana Documentation - Calling Between Programs - Hash-based generated program addresses
- docs.rs - solana_program::pubkey::find_program_address

ClaimableTokens - Derived address created in Claimable Tokens could be owned

Finding ID: KS-AUDPROT-09
Severity: **Informational**
Status: **Open**

Description

The derived address for the new account in `CreateTokenAccount` of the Claimable Tokens program could be a point on the ed25519 and thus have an associated private key. In general, a program derived address (PDA) should not lie on the curve. If a user happens to own the private key of the derived address, they could then sign for that account. The chances of this happening or abusing the above are negligible, since guessing the private key, or controlling instruction inputs in order to produce an address for a known private key, are considered impossible tasks.

Proof of issue

File name: solana-programs/claimable-tokens/program/src/utils/program.rs
Line number: 56

```
pub fn find_derived_address(
    base: &Pubkey,
    eth_public_key: EthereumAddress,
) -> Result<(Pubkey, String), PubkeyError> {
    let seed = bs58::encode(eth_public_key).into_string();
    Pubkey::create_with_seed(base, seed.as_str(), &spl_token::id()).map(|i|
(i, seed))
}
```

Severity and Impact summary

In an extreme scenario, if the derived public address corresponds to a user's private key, then that user would be able to sign for the account created.

Recommendation

To guarantee the PDA will reside outside the curve, you could use `find_program_address`, preferably off-chain, in order to calculate bump seeds and derive the address of the account, which could also be then created off-chain.

The on-chain program can then pass the bump seed and the additional seeds to `create_program_address` to calculate the account address for verification. The `create_program_address` function will fail if the calculated address lies on the ed25514 curve.

However, care should be taken with regards to off-chain calculation of the bump seed. Bump seeds should only be passed as part of initialization instructions. Otherwise, other bumps seed could be passed causing calculation of other valid addresses which could let an attacker bypass security checks.

RewardsManager - Token transfer from attestations can be stolen

Finding ID: KS-AUDPROT-10
Severity**: High**
Status: **Remediated**

Description

During attestation evaluation, the tokens are transferred to an account which is not related with the signed data. An attacker could listen on the Solana blockchain, until a set of valid attestations are generated. They could then use those attestations to send tokens to an arbitrary account.

Proof of Issue

File name: solana-programs/reward-manager/program/src/processor.rs
Line number: 469

```
        spl_token_transfer(
            program_id,
            &reward_manager_info.key,
            reward_token_source_info,
            reward_token_recipient_info,
            reward_manager_authority_info,
            transfer_data.amount,
        )?;
```

The address `reward_token_recipient_info` is not being used anywhere else in the function and could thus be selected arbitrarily.

Severity and Impact summary

An attacker could “steal” attestations and redirect attested transaction to arbitrary address.

Remediation

This issue was found in the given version of the code but after the initial of the draft report, after which the issue was already remediated. The solution derives `reward_token_recipient_info` from the program id, the ethereum address (which is signed), and the mint of `reward_token_source_info`. This links the signed data with the source and destination of the transaction.

METHODOLOGY

Kudelski Security uses the following high-level methodology when approaching engagements. They are broken up into the following phases.

Figure 2: Methodology Flow

Kickoff

The project is kicked off after the sales process has concluded. We typically set up a kickoff meeting where project stakeholders are gathered to discuss the project as well as the responsibilities of participants. During this meeting we verify the scope of the engagement and discuss the project activities. It's an opportunity for both sides to ask questions and get to know each other. By the end of the kickoff there is an understanding of the following:

- Designated points of contact
- Communication methods and frequency
- Shared documentation
- Code and/or any other artifacts necessary for project success
- Follow-up meeting schedule, such as a technical walkthrough
- Understanding of timeline and duration

Ramp-up

Ramp-up consists of the activities necessary to gain proficiency on the particular project. This can include the steps needed for familiarity with the codebase or technological innovation utilized. This may include, but is not limited to:

- Reviewing previous work in the area including academic papers
- Reviewing programming language constructs for specific languages
- Researching common flaws and recent technological advancements

Review

The review phase is where most of the work on the engagement is completed. This is the phase where we analyze the project for flaws and issues that impact the security posture. Depending on the project this

may include an analysis of the architecture, a review of the code, and a specification matching to match the architecture to the implemented code.

In this code audit, we performed the following tasks:

1. Security analysis and architecture review of the original protocol
2. Review of the code written for the project
3. Compliance of the code with the provided technical documentation

The review for this project was performed using manual methods and utilizing the experience of the reviewer. No dynamic testing was performed, only the use of custom-built scripts and tools were used to assist the reviewer during the testing. We discuss our methodology in more detail in the following sections.

Code Safety

We analyzed the provided code, checking for issues related to the following categories:

- General code safety and susceptibility to known issues
- Poor coding practices and unsafe behavior
- Leakage of secrets or other sensitive data through memory mismanagement
- Susceptibility to misuse and system errors
- Error management and logging

This list is general list and not comprehensive, meant only to give an understanding of the issues we are looking for.

Technical Specification Matching

We analyzed the provided documentation and checked that the code matches the specification. We checked for things such as:

- Proper implementation of the documented protocol phases
- Proper error handling
- Adherence to the protocol logical description

Reporting

Kudelski Security delivers a preliminary report in PDF format that contains an executive summary, technical details, and observations about the project.

The executive summary contains an overview of the engagement including the number of findings as well as a statement about our general risk assessment of the project as a whole. We may conclude that the overall risk is low but depending on what was assessed we may conclude that more scrutiny of the project is needed.

We not only report security issues identified but also informational findings for improvement categorized into several buckets:

- Critical
- High
- Medium
- Low
- Informational

The technical details are aimed more at developers, describing the issues, the severity ranking and recommendations for mitigation.

As we perform the audit, we may identify issues that aren't security related, but are general best practices and steps, that can be taken to lower the attack surface of the project. We will call those out as we encounter them and as time permits.

As an optional step, we can agree on the creation of a public report that can be shared and distributed with a larger audience.

Verify

After the preliminary findings have been delivered, this could be in the form of the approved communication channel or delivery of the draft report, we will verify any fixes withing a window of time specified in the project. After the fixes have been verified, we will change the status of the finding in the report from open to remediated.

The output of this phase will be a final report with any mitigated findings noted.

Additional Note

It is important to note that, although we did our best in our analysis, no code audit or assessment is a guarantee of the absence of flaws. Our effort was constrained by resource and time limits along with the scope of the agreement.

While assessment the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. These is a solid baseline for severity determination.

The Classification of identified problems and vulnerabilities

There are four severity levels of an identified security vulnerability.

Critical – vulnerability that will lead to loss of protected assets

- This is a vulnerability that would lead to immediate loss of protected assets
- The complexity to exploit is low
- The probability of exploit is high

High - A vulnerability that can lead to loss of protected assets

- All discrepancies found where there is a security claim made in the documentation that can not be found in the code
- All mismatches from the stated and actual functionality
- Unprotected key material
- Weak encryption of keys
- Badly generated key materials
- Tx signatures not verified
- Spending of funds through logic errors
- Calculation errors overflows and underflows

Medium - a vulnerability that hampers the uptime of the system or can lead to other problems

- Insecure calls to third party libraries
- Use of untested or nonstandard or non-peer-revied crypto functions
- Program crashes leaves core dumps or write sensitive data to log files

Low - Problems that have a security impact but does not directly impact the protected assets

- Overly complex functions
- Unchecked return values from 3rd party libraries that could alter the execution flow

Informational

- General recommendations

Tools

The following tools were used during this portion of the test. A link for more information about the tool is provided as well.

Tools used during the code review and assessment

- Rust – cargo tools
- IDE modules for Rust and analysis of source code
- Cargo audit which uses https://rustsec.org/advisories/ to find vulnerabilities cargo.

RustSec.org

About RustSec

The RustSec Advisory Database is a repository of security advisories filed against Rust crates published and maintained by the Rust Secure Code Working Group.

The RustSec Tool-set used in projects and CI/CD pipelines

'cargo-audit' - audit Cargo.lock files for crates with security vulnerabilities.

'cargo-deny' - audit Cargo.lock files for crates with security vulnerabilities, limit the usage of particular dependencies, their licenses, sources to download from, detect multiple versions of same packages in the dependency tree and more.

KUDELSKI SECURITY CONTACTS

NAME	POSITION	CONTACT INFORMATION