SECURITY AUDIT OF

XPLA FEE GRANT CONTRACT

Public Report

Dec 20, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
XPLA	XPLA is an open-source blockchain leveraging Tendermint and the Cosmos SDK. It offers extensive experiences in De-Fi and P2O gaming, aiming to transition Web2 users into the Web3 space with EVM compatibility and developer-friendly SDKs.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Dec 20, 2024. We would like to thank the XPLA for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the XPLA Fee Grant Contract. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team found no vulnerabilities in the given version of XPLA Fee Grant Contract, only some notes and recommendations.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About XPLA Fee Grant Contract

XPLA is a Tendermint-based Layer 1 blockchain that serves as a hub for digital media content. Based on the idea of 'Explore and Play', XPLA encompasses a wide range of digital content with a leading blockchain gaming infrastructure empowered by a sustainable ecosystem. As a universal content powerhouse, XPLA provides a sublime creative experience for all with its significant size game infrastructure.

1.2. Audit Scope

This audit focused on identifying security flaws in code and the design of the smart contracts of XPLA Fee Grant Contract.

It was conducted on the following folder from the git repository

https://github.com/xpladev/fee-grant-contract/

on commit `1ba43e8fdec64511b5dc8b66bdb6fac71d2559e3`.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
46d228dbccb62e1ab718e7053f88a45bf099935052e27985eb2505d0e9cbf1ed	./test.rs
7b19ef90785d6e310ac5cb4c7e42e4f60dfcbc9c0fc2c929968eaf831bd62665	./error.rs
d931692190c6c97472b46dc0413e1f8f984cf8ffcef0eecb3eef1275a8928d2d	./lib.rs
87f3b18a521b72a6ca1659e2c385c81bbadced035458e890e4620f1617665309	./state.rs
1cdc4e59077527941aa1968a86320046802145e46e23f5ebbd06d877c1112288	./contract.rs
f530d683e03a2f94af5eba7e985c00c439190181fc0d01c33b036fd03b13109e	./msg.rs

1.3. Audit Methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the CosmWasm smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- Gas Usage, Gas Limit and Loops
- Reentrancy
- Access Control
- Logic Flaws
- Use Of Unsafe Libraries
- Arbitrary Message Execution
- Bech32 Address Validation and Normalization

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

XPLA acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. XPLA understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, XPLA agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the XPLA will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the XPLA, the final report will be considered fully accepted by the XPLA without the signature.

2. AUDIT RESULT

2.1. Overview

XPLA Chain's fee grant module inherits from the Cosmos SDK's `feegrant` module. This module allows accounts to grant fee allowances and to use fees from their accounts. Grantees can execute any transaction without the need to maintain sufficient fees.

XPLA Fee Grant Contract is a fee grant contract written in `Rust` for the `XPLA` Chain. It allows the contract to grant fee allowances to grantees who can then use their allowance to pay for contract execution fee. The main functionalities include:

- **Instantiate**: The deployer initializes the contract with a `name` and `owner`.
- **Execute**:
  - `RegisterOperator` and `RemoveOperator`: Allows the `owner` to register and remove `operator`.
  - `Grant`: Allows the `owner` and `operator` to grant fee allowances to a list of `grantees`.
  - `Revoke`: Allows the `owner` and `operator` to revoke fee allowances from specific `grantees`.
- **Query**:
  - `AllGrantees`: Returns a paginated list of all grantees with fee allowances.
  - `IsGrantee`: Checks if a specific address is a registered grantee.
  - `OperatorInfo`: Returns the registered operator's details.
  - `Config`: Returns the contract configuration details.
- **Migrate**: Handles contract migration; currently not implemented yet.

2.2. Findings

During the audit process, the audit team found no vulnerabilities in the given version of XPLA Fee Grant Contract, only some notes and recommendations.

#	Issue	Severity	Status
1	Duplicated logic code	INFORMATIVE	Open
2	Scoped approval is not supported	INFORMATIVE	Open

2.2.1. Duplicated logic code INFORMATIVE

Affected files:

- contract.rs

In both `_grant` and `revoke` functions, there are duplicated logic for:

- Authorization checks
- Empty grantees validation
- Max limit validation
- Filter unique and validate grantees addresses

Consider moving the duplicated logic into a helper function to share the logic for both `_grant` and `revoke` functions. This will make the code more maintainable and easier to read, avoid future bugs when the logic is updated.

```
fn _grant(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
    grantees: Vec<String>,
) -> Result<Vec<CosmosMsg>, ContractError> {

    let operator = OPERATOR.load(deps.storage).ok();
    let config = CONFIG.load(deps.storage)?;

    //Duplicated logic.
    if operator.as_ref().map_or(true, |op| op.operator != info.sender) && config.owner !=
info.sender {
        return Err(ContractError::Unauthorized {});
    }

    if grantees.is_empty() {
        return Err(ContractError::EmptyGrantees {});
    }
    if grantees.len() > MAX_LIMIT_MSG {
        return Err(ContractError::TooManyGrantees {max:MAX_LIMIT_MSG});
    }

    let unique_grantees: HashSet<_> = grantees.clone().into_iter().collect();
    for grantee in &unique_grantees {
        let grantee = deps.api.addr_validate(grantee)?;
        if GRANTEES.has(deps.storage, grantee.clone()) {
            return Err(ContractError::AlreadyAddGrantees {});
        }
    }
    ____________________________________________________________________________
}

pub fn revoke(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
    grantees: Vec<String>,
) -> Result<Response, ContractError> {
    let operator = OPERATOR.load(deps.storage).ok();
    let config = CONFIG.load(deps.storage)?;

    //Duplicated logic.
    if operator.map_or(true, |op| op.operator != info.sender) && config.owner !=
info.sender {
        return Err(ContractError::Unauthorized {});
    }

    if grantees.is_empty() {
        return Err(ContractError::EmptyGrantees {});
    }
    if grantees.len() > MAX_LIMIT_MSG {
        return Err(ContractError::TooManyGrantees {max:MAX_LIMIT_MSG});
    }

    let unique_grantees: HashSet<_> = grantees.clone().into_iter().collect();
    for grantee in &unique_grantees {
        let grantee = deps.api.addr_validate(grantee)?;
        if !GRANTEES.has(deps.storage, grantee.clone()) {
            return Err(ContractError::GranteeNotAdded {});
        }
    }
    ________________________________________________________________________
}
```

In `remove_operator` function, the logic for removing `operator` and `add_attribute` are duplicated as well.

```
pub fn remove_operator(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
) -> Result<Response, ContractError> {

    let config = CONFIG.load(deps.storage)?;
    if config.owner != info.sender {
        return Err(ContractError::Unauthorized {});
    }

    let op = OPERATOR.load(deps.storage)?;
    let operator_addr = op.operator.clone();
    if GRANTEES.has(deps.storage, operator_addr.clone()) {
        let granter_contract = env.contract.address;

        let msg = Anybuf::new()
            .append_string(1, granter_contract.clone())
            .append_string(2, operator_addr.to_string());

        let revoke_message = CosmosMsg::Stargate {
            type_url: "/cosmos.feegrant.v1beta1.MsgRevokeAllowance".to_string(),
            value: msg.into_vec().into(),
        };

        GRANTEES.remove(deps.storage, operator_addr.clone());
        OPERATOR.remove(deps.storage);

        return Ok(Response::new()
            .add_message(revoke_message)
            .add_attribute("action", "remove_operator")
            .add_attribute("operator", operator_addr)
        );
    }

    OPERATOR.remove(deps.storage);
    Ok(Response::new()
        .add_attribute("action", "remove_operator")
        .add_attribute("operator", operator_addr)
    )
}
```

We can move the logic for removing `operator` and `add_attribute` out of if condition. We only need to remove grantee and add `revoke_message` to the response if the operator is a grantee.

RECOMMENDATION

Refactor the code to avoid duplicating logic. This will make the code more maintainable, easier to read, and help prevent future bugs by ensuring that updates to the logic are made in a single place.

2.2.2. Insufficient Test Cases INFORMATIVE

The current contract includes only a few basic unit tests. It is advisable to expand the test suite to cover all functionalities of the contract comprehensively. This should include integration tests that interact with the blockchain.

Doing so will help ensure that the contract operates as intended and that `grantees` can use their allowances to pay for contract execution fees.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Dec 20, 2024*	Public Report	Verichains Lab

Table 2. Report versions history