Cyberscope

Audit Report

ATLETA

September 2024

Repository	https://github.com/Atleta-network/atleta/tree/devnet/pallets/faucet
Commit	8afb82d543718879163f8f3b1e26e986d17d2407
Audited by	© cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/Atleta-network/atleta/tree/devnet/pallets/faucet
Commit	8afb82d543718879163f8f3b1e26e986d17d2407
Network	Atleta

Audit Updates

Initial Audit	18 Sept 2024

Source Files

Filename	SHA256
src/benchmarking.rs	7a67fde580d7ef6c76cf8a284f009a684350ec1f85dd6fa66a3a65ec01531706
src/lib.rs	f8702c496fe6193cf0a70479b511ba58a3d7e74fd77af6d61abe305c04adfa53
src/mock.rs	4acdee5888057b926c7da67b9cb161f2daa898e9f0ae31e1f50083213a7d49a4
src/tests.rs	c941aec8fb4353a07d7b17d3e219adb73a81410e34ceb163c7afdb9c469d1470
src/weights.rs	79884efab750f4594b7c0faf2d753159fc54ecb6dc6d25a905554803de553c94

Overview

The `lib.rs` file in the `pallet_faucet` module implements a faucet mechanism for distributing tokens on a Substrate-based blockchain. The primary functionality of the pallet is to allow users to request a specified amount of tokens for their own accounts, within predefined limits. Users are restricted to requesting a maximum amount, defined by the `Config::FaucetAmount` parameter, within a given time period specified by `Config::AccumulationPeriod`. The pallet utilizes the `Currency` trait to manage token balances and transfers, ensuring that the requested amount is transferred from the faucet's account to the user's account.

The pallet enforces limits on requests to prevent abuse by checking the requested amount against the maximum allowed and ensuring that the cumulative amount requested during the defined period does not exceed the specified threshold. If these conditions are met, the pallet updates its internal storage to track the request and emits an event indicating that the funds have been sent. The `Config::PalletId` is used to derive a unique account ID for the faucet, and this account's balance is managed to ensure it has sufficient funds for distribution. Additionally, the pallet includes a genesis configuration to initialize the faucet's account with a minimum balance at the genesis block. The pallet is designed for use in test networks, providing a controlled environment for distributing tokens to users for testing purposes.

Tests and Benchmarks Overview

The `pallet_faucet` module includes a set of tests and benchmarks designed to validate and measure the performance of the faucet's functionality. The tests ensure that the key behaviors of the faucet, such as requesting funds and handling errors, are implemented correctly, while the benchmarks provide insights into the computational cost of executing these operations.

Tests

The unit tests are defined in the `mock.rs` and `tests.rs` files. They simulate different scenarios to validate the behavior of the `request_funds` function. The tests cover various cases, including successful fund requests, failure due to exceeding the maximum allowed amount, and failure when the cumulative requests within a specific period exceed the configured limit. The mock runtime environment is configured with custom parameters, such as `FaucetAmount` and `AccumulationPeriod`, and uses the `ExtBuilder` utility to build and execute the test environment. Assertions are made using macros like `assert_ok!` and `assert_noop!` to ensure that the faucet behaves as expected under different conditions.

For instance, the `faucet_works` test checks that a user can successfully request a specified amount of funds, and the faucet account is correctly debited. Other tests, such as `faucet_fail_send_more_than_max` and `faucet_fail_exceed_max_amount_during_period`, verify that appropriate errors are thrown when the request exceeds the maximum allowable amount or when the accumulation period is violated.

Benchmarks

The benchmarks are defined in the `benchmarking.rs` file and are used to measure the execution time and computational resources required by the `request_funds` function. The `frame_benchmarking::v2` framework is used to set up and execute these benchmarks. The `request_funds` benchmark simulates a typical fund request operation, and the performance metrics are captured and stored in a `weights.rs` file. This file contains auto-generated weights for the faucet operations, providing data for tuning the pallet's performance in the runtime environment.

The benchmarking suite ensures that the `request_funds` function operates efficiently under various conditions. By running the benchmarks, insights can be gained of the operational costs associated with the faucet’s functionality. The results are used to adjust the weights associated with the pallet’s functions, ensuring that they reflect the actual computational effort required, which is crucial for maintaining optimal performance and resource allocation in the blockchain runtime.

Findings Breakdown

Critical	0
Medium	1
Minor / Informative	3

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	1	0	0	0
Minor / Informative	3	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	VDU	Vulnerable Dependencies Usage	Unresolved
Minor / Informative	IOCI	Inconsistent Origin Check Implementation	Unresolved
Minor / Informative	UEH	Unchecked Error Handling	Unresolved
Minor / Informative	IBM	Inefficient Balance Management	Unresolved

VDU - Vulnerable Dependencies Usage

Criticality	Medium
Status	Unresolved

Description

The project relies on several third-party crates that contain known security vulnerabilities or have been marked as unmaintained. These vulnerabilities could expose the project to security risks, including timing attacks, denial of service, infinite loops, and configuration corruption. Additionally, using unmaintained dependencies can introduce further risks, as they may no longer receive security patches or updates. These issues affect critical crates within the dependency tree, potentially impacting the overall security and stability of the project.

Errors:

- Timing variability in `curve25519-dalek`'s `Scalar29::sub` / `Scalar52::sub`
- Double Public Key Signing Function Oracle Attack on `ed25519-dalek`
- `MemBio::get_buf` has undefined behavior with empty buffers
- `rustls::ConnectionCommon::complete_io` could fall into an infinite loop based on network input
- Binary Protocol Misinterpretation caused by Truncating or Overflowing Casts

Warnings:

- unmaintained: `ansi_term` : 0.12.1, `mach` : 0.3.2, `parity-wasm` : 0.45.0, `proc-macro-error` : 1.0.4
- yanked: `bytemuck` : 1.16.1 , `bytes` : 1.6.0

Recommendation

Review all identified vulnerabilities and unmaintained dependencies in the project's third-party crates. It is recommended to update the affected crates to the latest secure versions where possible, or consider alternative libraries that are actively maintained. Regularly monitoring and auditing third-party dependencies is crucial to ensure the project's security and integrity, reducing exposure to potential risks.

IOCI - Inconsistent Origin Check Implementation

Criticality	Minor / Informative
Location	src/lib.rs#L2,128
Status	Unresolved

Description

The `request_funds` function currently supports unsigned calls, allowing any user or process to request funds without authentication. This is confirmed by the use of an origin check that ensures the origin is `None` and the implementation of a mechanism to validate unsigned extrinsics. However, the comment above the function states that the origin should be signed, which suggests that only authenticated users are intended to call this function. This creates confusion and inconsistency between the documented purpose and the actual behavior of the function.

```
//! The origin should be signed.

pub fn request_funds(
            origin: OriginFor<T>,
            who: T::AccountId,
            amount: BalanceOf<T>,
        ) -> DispatchResult {
            ensure_none(origin)?;

            ensure!(amount <= T::FaucetAmount::get(),
Error::<T>::AmountTooHigh);

            let (balance, timestamp) = Requests::<T>::get(&who);
            let now = frame_system::Pallet::<T>::block_number();
            let period = now - timestamp;

            let (total, now) = if period >=
T::AccumulationPeriod::get() {
                (amount, now)
            } else {
                (balance + amount, timestamp)
            };

            ensure!(total <= T::FaucetAmount::get(),
Error::<T>::RequestLimitExceeded);

            let account_id = Self::account_id();

            let _ = T::Currency::make_free_balance_be(&account_id,
amount);
            let _ =
                T::Currency::transfer(&account_id, &who, amount,
ExistenceRequirement::AllowDeath);

            Requests::<T>::insert(&who, (total, now));

            Self::deposit_event(Event::FundsSent { who, amount });

            Ok(())
        }
    }

    #[pallet::validate_unsigned]
    impl<T: Config> ValidateUnsigned for Pallet<T> {
        type Call = Call<T>;

        fn validate_unsigned(_source: TransactionSource, call:
&Self::Call) -> TransactionValidity {
            match call {
                Call::request_funds { who, amount } =>
```

```
ValidTransaction::with_tag_prefix("Faucet")
                        .and_provides((who, amount))
                        .propagate(true)
                        .build(),
                    _ => InvalidTransaction::Call.into(),
                }
            }
        }
```

Recommendation

To resolve this inconsistency, the intended use of the `request_funds` function should be clearly defined. If unsigned calls are acceptable and intended, the comments should be updated to reflect this, providing clarity on the function's purpose and the controls in place to prevent misuse. On the other hand, if only authenticated users are permitted to request funds, the origin check should be modified to ensure that the function only accepts signed calls, and the unsigned validation mechanism should be adjusted accordingly. This will align the implementation with the intended security model and prevent potential misuse or confusion.

UEH - Unchecked Error Handling

Criticality	Minor / Informative
Location	src/lib.rs#L151,152
Status	Unresolved

Description

The `request_funds` function uses expressions that discard potential errors during balance updates and transfers. This is done by ignoring the return value of critical operations, such as updating the faucet account's balance or transferring tokens to the user's account. Ignoring these results can lead to silent failures, where issues such as insufficient funds or failed transfers are not detected, and the contract execution proceeds as if everything succeeded. This could result in users not receiving their requested funds while the contract still appears to function normally.

```
let _ = T::Currency::make_free_balance_be(&account_id, amount);
let _ =
    T::Currency::transfer(&account_id, &who, amount,
ExistenceRequirement::AllowDeath);
```

Recommendation

It is recommended to properly handle the return values of balance update and transfer operations. Ensure that any errors encountered during these operations are logged or cause the transaction to fail gracefully. This approach will improve the reliability of the contract by preventing silent failures and providing better transparency and traceability for any issues that arise.

IBM - Inefficient Balance Management

Criticality	Minor / Informative
Location	src/lib.rs#L151
Status	Unresolved

Description

The `request_funds` function performs an unnecessary sequence of operations by first updating the faucet account's balance using a direct balance-setting method and then transferring the amount to the recipient. This approach introduces redundancy and potential risks, as directly setting the balance can overwrite existing values without proper checks. The transfer method, which includes built-in validations, is sufficient for moving funds between accounts and helps maintain the integrity of the balance management. Using both methods in sequence is not only inefficient but also increases the complexity of the code.

```
let _ = T::Currency::make_free_balance_be(&account_id, amount);
let _ =
    T::Currency::transfer(&account_id, &who, amount,
ExistenceRequirement::AllowDeath);
```

Recommendation

The function should be optimized by removing the direct balance update operation and relying solely on the transfer method to move funds from the faucet account to the recipient. This simplifies the logic and ensures that balance changes occur through validated and controlled operations. If direct balance modifications are necessary in specific scenarios, adequate checks should be in place to prevent overwriting or unexpected alterations to account balances.

Summary

The `pallet_faucet` module implements a faucet mechanism for the Atleta Network. This audit investigates security issues, business logic concerns, and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope’s prior written consent. This report is not nor should be considered an “endorsement” or “disapproval” of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope’s position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io