QuakeCash Audit Report

Reviewed by:

- ParthMandale
- LordAlive

Prepared For: QuakeCash Protocol

Review Date(s): 23/7/25 - 30/7/25

About Security Researcher(s)

Audit is led by two experienced security researchers with over 1.5 years in the smart contract security space, with a proven track record. We've collectively uncovered 100+ H/M severity vulnerabilities across top-tier protocols. Our experience spans both public contests and private audits, with a strong emphasis on deep technical analysis, precise reporting, and validated findings. We're committed to delivering high-impact results in every engagement.

Protocol Summary

Quake Cash is a decentralized, gamified yield protocol where users deposit stablecoins to earn high yield, per-second interest, with withdrawals unlockable after a short delay and risks.

Scope

Repo: quake-cash-contracts - @ c16b35e7aa2276bfd4e4a2df2dbb6c12ee0dc5da

Repo: quake-cash-trigger-contracts - @ 8bfc0a17cd133cd560c0ab102f562009d0d35443

Deployment Chain(s):

- Base

Summary of Findings

Identifier	Title	Severity	Mitigated
[H-01]	Inaccurate Total Yield Calculation in `_currentDeposit()` Allows Epoch Deposit Limits to Be Exceeded	HIGH	Fixed
[H-02]	User's getting yield even after expiration time will lead to loss of funds to the protocol	HIGH	Partially Fixed
[M-01]	Change in `delayTimeSeconds` Can Lead to Unforeseen Loss of User Funds	MEDIUM	Fixed
[M-02]	Self-Referral Exploit Allows Users to Claim Unintended Bonus Yiel	MEDIUM	Fixed
[M-03]	Hardcoded expectedPrefix in `triggerEvent` can have adverse effects	MEDIUM	Fixed

	on the protocol		
[M-04]	Front-Running Vulnerability Due to Latency in Manual `triggerEvent` function Call	MEDIUM	Acknowledged
[L-01]	Ineffective Referral Code Boost Due to incorrect Rate Index	LOW	Fixed
[L-02]	Incorrect Comparison to String Literal in `_getIntegerFromString`	LOW	Fixed
[L-03]	Edge Case in getDepositBalance Fails to Nuke Deposit	MEDIUM	Fixed
[L-04]	Insufficient Rate Indices while setting `_setRates` Can Break Donation and Referral Logic	LOW	Acknowledged
[I-01]	Input params should be checked with require and not with assert	INFORMATIONAL	Fixed
[I-02]	Incorrect NatSpec Comments in QuakeCash Contract	INFORMATIONAL	Fixed

---

Findings

[H-01] Inaccurate Total Yield Calculation in `_currentDeposit()` Allows Epoch Deposit Limits to Be Exceeded.

Description

The contract's mechanism for enforcing the deposit limit per epoch is flawed. The `deposit()` function checks the current Global Deposit and its total yeild against the `depositLimit` for the current epoch through the function `_currentDeposit`. However, the `_currentDeposit` function, which calculates this current Global Deposit and its yeild accrued till current time, incorrectly determines the total yield generated.

The current implementation only calculates the yield based on the `baseRateIndex`, which includes the base yield and any yield from donations. It completely omits the additional yield generated from referral bonuses.

As a result, the contract underestimates the true total value of deposits plus accrued interest. This allows new deposits to be accepted even after the actual real yield value has surpassed the epoch's `depositLimit`.

```
// function deposit()
@1> require(_currentDeposit() <= latestEpoch.depositLimit, "Deposit exceeds global
limit");

// function _tryToggleCompoundAndLinear()
```

```
@2> if (_currentDeposit() >= latestEpoch.depositLimit) // switch to linear interest
yeild generation



    function _currentDeposit() internal view returns (uint256) {
        Epoch memory latestEpoch = epochs[epochs.length - 1];
        uint16 baseRateIndex = rateSets[latestEpoch.rateSetIndex].baseteIndex;
@3>     LiquidityIndex memory liquidityInfo =
latestEpoch.liquidityIndices[baseRateIndex];

        if (liquidityInfo.totalScaled == 0) return 0;

        // Convert scaled value using last-known liquidity index (RAY precision → 6
decimals)
        return _denormalizeAmount(liquidityInfo.totalScaled,
liquidityInfo.liquidityIndex);
    }
```

Impact

This bug has several bad impacts:

- Loss of Funds for Yield Contract(owner): The yield contract will be forced to pay significantly more in interest than was budgeted for the deposit limit (as explained above, this is happening because even after the real `depositLimit` will be reached, but still deposit function will allow taking more deposit), leading to a direct financial loss for the protocol owner. The excess deposits will also generate a high daily yield (e.g., 30%), compounding the financial drain on the yield contract.
- Breaks Core Invariant: The primary rule that total deposits plus yield total yeild generated on it cannot exceed the epoch's limit is violated, undermining the contract's economic stability.
- Incorrect Switch between Linear Interest rate and Compound interest rate will happen as the `_tryToggleCompoundAndLinear` function will also work on the same concept.

Recommendation

Function `_currentDeposit()` & `currentDeposit()` should correctly calculate the yeild of all deposits that include base index yield(donation yield gets covered into it) + referral index yield for that epoch.

[H-02] User's getting yield even after expiration time will lead to loss of funds to the protocol

Description

Users are still receiving yield even after the expiration time (`expireTime`) has passed. This creates a critical exploit vector:

- A malicious user can set the `expireTime` to a minimal value, such as 10 minutes.
- After this period ends, instead of calling `_processDeposit`, the user passively monitors the mempool.
- When a `triggerEvent` is detected, indicating a possible yield distribution, the user **front-runs** the `_processDeposit` transaction.
- This allows them to **claim yield without taking any risk** during the actual yield-generating period.

```
if (
            // Deposit expired before now AND
            // Deposit made in most recent epoch (no events occurred since)
            // OR: deposit expired before next event
            userDeposit.expireTime > 0 && userDeposit.expireTime < block.timestamp
                && (
                    (userDeposit.epochIndex == events.length)
                        || (userDeposit.expireTime <
events[userDeposit.epochIndex].reportedAt)
                )
        ) {
            // *** CLAIMING ***
            userDeposit.claimedTime = uint64(block.timestamp);

            // Calculate depositor's max earnable balance using base rate
            uint256 depositorBalance = _denormalizeAmount(
                userDeposit.scaledDeposit,
epochs[userDeposit.epochIndex].liquidityIndices[baseRateIndex].liquidityIndex
            );
```

The core issue arises due to the protocol still giving out the **Yield** even after completion of `expireTime`. If left unaddressed, the exploit could be automated via bots, scaled across multiple wallets, and significantly harm the protocol's financial integrity and trust.

Proof of Concept (PoC)

```
    function test_deposit_at_epoch_end_full_yield_2() public {
        // Setup user and get initial balance
        address user = setupUser(0);
        uint256 balanceBefore = stableToken.balanceOf(user);
        uint128 amount = 100e6; // 100 USDC

        // Define epoch and deposit timing variables
        uint256 epochDuration = SECONDS_PER_DAY;
        uint64 minDepositDuration = quakeCash.delayTimeSeconds(); // 10 minutes

        // User deposits with the minimum expiry
        quakeCash.deposit(amount, 0, minDepositDuration, 0);

        // full 1 day epoch time done
        vm.warp(block.timestamp + epochDuration);
```

```
        // Get the expected balance from the contract itself
        uint256 expectedBalance = quakeCash.getDepositBalance(user, 0);

        // successfull Process the deposit without revert will happen.
        _processDeposit(user, 0, 1);

        // Assert that the user received full 1 day yeild where as he only took the
minimum expiry time (10 minutes) of risk.
        // 130e6 -> some wei are rounded down, making it 129.999948e6
        assertEq(129999948, expectedBalance, "Incorrect yield received");
    }
```

Impact

- Users can extract unfair yield with **zero exposure to protocol risk**.
- Potential for **mass exploitation** through bots or automated systems that detect yield-trigger events.

Recommendation

- Store user addresses in an array and make a view function to get the user addresses and indices whose expiryTime is completed and automate `_processDeposit` function with that. **Below are some alternate ways to fix the issue , But a bit complex**:- - Stop rewarding users after the `expiryTime` ends - Add another waiting period at the time of `_processDeposit`, so that the users cannot perform instantaneous actions which leads to the exploit.

[M-01] Change in `delayTimeSeconds` Can Lead to Unforeseen Loss of User Funds

Description

The contract allows a privileged role to change the global state variable `delayTimeSeconds` through the function `_setDelayTimeSeconds`, which is a crucial time window that users with `expiryTime` = `0` have to go through when they want to instantly withdraw their deposits through function `_initiateWithdrawal`.

The vulnerability lies in the fact that the withdrawal process (`_initiateWithdraw`) uses the current, global state variable `delayTimeSeconds` to determine the cooling period after which only the users can claim their yeild through the function `processDeposits`. This means if an admin increases the delay from, for example, 10 minutes to 2 hours, all existing deposits are immediately subject to this new, much longer risk window.

This fundamentally alters the terms of the deposit after the fact and without the user's consent. It particularly harms users who deposited with expireTime = 0, as they likely planned for an instant withdrawal strategy based on the shorter, initial delay period.

Impact

- The risk profile of a user's deposit can be changed arbitrarily by an admin, breaking the implicit agreement made at the time of deposit.

- Users who planned their strategy around a short delay window could lose their entire deposit if an earthquake occurs during the newly extended, unanticipated lock-up period.

Recommendation

To ensure that the risk parameters are locked in at the time of deposit, the following changes can be made:

Modify the Deposit Struct: Add a new field to the Deposit struct to store the `delayTimeSeconds` value and modify this at the time of the user's deposit, and at the time of `_initiateWithdraw` of the user, the same `delayTimeSeconds` attached to the Deposit struct of the user can be used.

[M-02] Self-Referral Exploit Allows Users to Claim Unintended Bonus Yield

Description

The contract's referral system can be exploited by users to earn bonus yield on their own deposits. The function `deposit` uses this check `require(referralCode < nextCode, "Invalid referral code");` in order to prevent the user to not predicting their own future referral code and use it to deposit on their own deposit.

But the issue can still be exploited by a malicious user through the following steps:

- Make a very small initial deposit (e.g., 1 wei) without a referral code. This action generates a personal referral code for their address.
- Make a second, much larger deposit, but this time, use their own newly acquired referral code.

Because the contract does not check if the depositor and the referrer are the same entity, the user successfully earns the referral bonus yield on their own capital. This circumvents the intended purpose of the referral program, which is to reward users for bringing new participants and capital into the protocol.

Impact

Malicious users can claim extra yield from the protocol that was meant to incentivize genuine protocol community growth through a referral system. It ceases to be a marketing tool and becomes a loophole for sophisticated users to inflate their own returns. The exploit is significantly worse if an admin applies a rate boost to the user's referral code. The user can then generate an even larger(by depositing larger), unearned yield for themselves at the protocol's yeild expense.

Recommendation

The fix is to add a simple check in the deposit function. When a deposit is made with a referral code, the contract must verify that the depositor is not also the owner of that referral code. This check should be implemented from the second deposit of the user.

[M-03] Hardcoded `expectedPrefix` in triggerEvent can have adverse effects on the protocol

Description

The `EventTrigger.sol::triggerEvent` function uses a hardcoded value called `expectedPrefix`, which points to a centralized website that provides information about earthquakes. This hardcoded URL serves as the sole source for validating and triggering earthquake-related events in the contract.

However, relying exclusively on a centralized web source introduces a significant single point of failure. If the specified website experiences temporary downtime during an actual earthquake, the event cannot be reported or processed on-chain because no alternative sources are accepted. This design limitation could lead to critical real-world events being ignored by the protocol, affecting any downstream systems or automations dependent on timely disaster reporting.

Worse yet, if the centralized website permanently shuts down or changes its domain structure, the smart contract will effectively become non-functional for its core purpose. In such a case, the entire protocol may be forced to halt operations due to the inability to verify and register new earthquake events, demonstrating a fragility that is unacceptable in a decentralized system. Relying on a mutable, external, and centralized data source violates best practices for smart contract design and severely undermines the protocol's reliability and resilience.

```
    /// @param proof Reclaim proof with relevant API data
    function triggerEvent(Reclaim.Proof memory proof) external {
        // Check context for valid data
        Reclaim(reclaimAddress).verifyProof(proof);

        string memory magnitude =
Claims.extractFieldFromContext(proof.claimInfo.context, '"mag":"');
        string memory id = Claims.extractFieldFromContext(proof.claimInfo.context,
'"id":"');
        string memory timestamp =
Claims.extractFieldFromContext(proof.claimInfo.context, '"time":"');

        // Check url matches expected source
        string memory url = Claims.extractFieldFromContext(proof.claimInfo.parameters,
'"url":"');
@>      string memory expectedPrefix =
"https://earthquake.usgs.gov/fdsnws/event/1/query?format=geojson&eventid=";
        string memory expectedUrl = string(abi.encodePacked(expectedPrefix, id));

        require(keccak256(bytes(url)) == keccak256(bytes(expectedUrl)), "Invalid data
source");

        TargetTrigger(targetAddress).triggerEvent(magnitude, id, timestamp);
    }
```

Impact

The use of a centralized and hardcoded `expectedPrefix` introduces a **critical single point of failure** into the protocol. If the earthquake information website goes offline temporarily or permanently, the contract will be incapable of recording future earthquake events. This dependency not only halts the contract's core functionality but also undermines the decentralization and trustlessness expected in blockchain-based systems. In the worst-case scenario, the entire protocol could become non-operational, resulting in **system-wide disruption and loss of utility** for all dependent applications and users.

Recommendation

- Do not hardcode the `expectedPrefix` value instead, make a function to set it so that it can be modified whenever needed.

[M-04] Front-Running Vulnerability Due to Latency in Manual `triggerEvent` function Call

Description

The contract's security model relies on the `triggerEvent` function being called to log a real-world seismic event. However, this function must be called manually, creating a critical delay between the actual event and the contract's awareness of it.

This latency can be exploited by a malicious or sophisticated user. Such a user can monitor official earthquake data sources (e.g., USGS) and the Reclaim protocol's mempool transactions registry in real-time. Upon detecting a new, valid event, they can execute the transaction to withdraw their own funds by calling the function `_initiateWithdrawal` and completing with 10-minute cooldown delay time, before the `triggerEvent` function is successfully called for that same event.

This is especially problematic for deposits made with expireTime = 0, which are designed for quick withdrawal. The user can safely initiate their withdrawal, effectively avoiding the "nuking" penalty, and later process the deposit to claim their full yield. They get benefit from the high-yield returns without ever being exposed to the corresponding risk, which undermines the core economic principle of the protocol.

Impact

- Unfair Advantage: Creates a two-tiered system where technical/malicious users with bots can systematically outperform regular users by avoiding risk.
- Potential for Yield Depletion: The yield contract pays out funds to users who should have had their deposits "nuked," leading to a slow drain of protocol funds.

Recommendation

The triggerEvent function must be called as quickly as possible after a real-world event. Relying on a single party or altruism is not robust. A decentralized oracle mechanism or a decentralized incentive mechanism should be implemented.

[L-01] Ineffective Referral Code Boost Due to incorrect Rate Index

Description

When setting a referral code's boost to the index that is less than or equal to (BaseRateIndex + 1) referrer gains no advantage because the `referralOffset` is fixed at (BaseRateIndex + 1). This means the effective rate will remain the same, making the boost useless and potentially misleading for referrers.

Recommendation

Enforce a check to have a minimum boost index that is greater than (BaseRateIndex + 1) when setting up referral code rates. This ensures referrers receive a meaningful increase in their yield, maintaining the incentive structure and preventing confusion.

[L-02] Incorrect Comparison to String Literal in `_getIntegerFromString`

Description

In the function `_getIntegerFromString`, the following line attempts to compare a single byte from a `bytes` array to a string literal:

```
 function _getIntegerFromString(string memory input) internal returns (uint256 x) {
        bytes memory inputBytes = bytes(input);
        uint256 length = inputBytes.length;

        require(length > 0, "Input cannot be empty");

        x = 0;
        bool foundDecimal = false;

        for (uint256 i = 0; i < length; i++) {
            // Stop parsing when decimal point is found
@>            if (inputBytes[i] == ".") {
                foundDecimal = true;
                break;
            }
```

This comparison is **incorrect** because:

- `inputBytes[i]` is of type `bytes1`
- `"."` is a string literal of type `string memory`

In Solidity, comparing a `bytes1` to a `string` type is **invalid** and will result in an unintended behaviour.

Recommendation

- Use the ASCII byte representation of the dot (.) character, which is 0x2E for a proper byte-to-byte comparison.

[L-03] Edge Case in getDepositBalance Fails to Nuke Deposit

Description

The `getDepositBalance` function incorrectly uses a strict `>` comparison when checking if a deposit is nuked. If a deposit's expireTime is exactly equal to the event's reportedAt timestamp, it should be considered nuked, but the current logic will not return zero.

Recommendation

apply correct check in `getDepositBalance` function - `d.expireTime >= events[d.epochIndex].reportedAt`

[L-04] Insufficient Rate Indices while setting `setRates` Can Break Donation and Referral Logic

Description

For the protocol to function as intended, there must always be at least one rate index before and after the base rate index. This is necessary for correct donation and referral offset calculations. Additionally, a third index is also required for boost features to work properly.

Recommendation

Enforce a minimum number of rate indices checks and proper positioning relative to the base rate index when configuring rates, ensuring all protocol features operate as designed.

[I-01] Input params should be checked with require and not with assert

Description

- assert is used to catch internal bugs or invariants that should never be false.
- It’s not meant for input validation — for that, we should use require.
- All gas is consumed when tx fails(i.e., no refund of unused gas).

Function `_setRatesInternal` Input validations are done through assert, whereas these input params should be checked through `require`

```
        // At least one rate must be provided
        assert(_validRates.length > 0);

        // Base rate index must fall within valid rates array
        assert(baseRateIndex < _validRates.length);
```

Recommendation

- Use revert to check these inputs

[I - 02] Incorrect NatSpec Comments in QuakeCash Contract

Description

- Correct the NatSpec comments for the `_getIntegerFromString` function by replacing "@param" to "@return" for: "x The integer value of the parsed string"
- `currentGlobalDepositLimit` function's @return comment can be improved by replacing it with "Max total deposit + yield" same as `depositLimit` for any epoch.

Recommendation

Correct the Natspec Comments