PUBLIC

Code Assessment

of the Quark V2 and Quark Scripts Smart Contracts

31 October, 2024

Produced for

LEGEND

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Legend team,

Thank you for trusting us to help Legend Labs with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Quark V2 and Quark Scripts according to Scope to support you in forming an opinion on their security risks.

Legend Labs implements Quark v2, a smart contract wallet that enables accounts to run arbitrary scripts, Legend Labs also provides a suite of scripts to facilitate wallet operation and interact with DeFi systems. This audit follows our first audit of Quark, which can be found here. The new system implements an updated version of nonce control and state isolation, and introduces transient storage.

The most critical subjects covered in our audit are callback handling, nested operations, nonce processing, and slippage protection in swaps. Security regarding all aforementioned subjects is high. The unexpected slippage caused by accumulation of deviations of oracles, described in RecurringSwap Oracle deviations contributing to slippage, has been acknowledged as part of the behavior of the system, and properly documented.

All the issues raised have been satisfactorily addressed by Legend Labs, however a QuarkWallet is designed to execute arbitrary code in the context of a user's wallet through delegatecall. Script developers must understand the core mechanics of the Quark wallet before integrating with it, and Legend Labs should safeguard users against blind-signing malicious payloads by providing appropriate tooling to inspect wallet operations.

In summary, we find that the Quark codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	4
• Code Corrected	3
• Specification Changed	1
Low-Severity Findings	1
• Code Corrected	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Quark V2 and Quark Scripts repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

quark

V	Date	Commit Hash	Note
1	16 September 2024	622f8b560c5a3826c2b6c37f31d0937393488e14	Quark-v2 Initial
2	21 October 2024	dbbc36574bed0191b91ffc8ada94d24377db5e47	Quark-v2 fixes

quark-scripts

V	Date	Commit Hash	Note
1	16 September 2024	36d8a877ea5e6b8a3a5b044a1d75a61463777f5b	Initial Version
2	22 October 2024	6d738f60d61d8082c83c917dee5aff3995e3dca3	Quark scripts fixes
3	29 October 2024	24617d65b06937a25ff78b77b555dff9da41e649	Version 3

For the solidity smart contracts, the compiler version `0.8.27` was chosen.

The contracts included in the following files are in the scope of the review:

`quark`:

- quark-core/src/QuarkWallet.sol
- quark-core/src/QuarkNonceManager.sol
- quark-core/src/QuarkScript.sol

`quark-scripts`:

- src/Cancel.sol
- src/MorphoScript.sol
- src/RecurringSwap.sol
- src/UniswapFlashLoan.sol
- src/UniswapFlashSwapExactOut.sol

2.1.1 *Excluded from scope*

Tests and files not explicitly listed above are excluded from scope. Moreover third party libraries are assumed to behave correctly according to their specification.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

At the end of this report section, we have added subsections for each of the changes according to the versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Legend Labs provides the "Quark v2" smart contract wallet system.

2.2.1 QuarkWallet

The `QuarkWallet` has a `signer` and `executor` who can trigger the execution of external scripts. Scripts are contracts that may contain arbitrary code and are deployed by the Quark system's `CodeJar` contract.

The `executor` of the wallet has a privileged role and can directly call the `executeScript` function with valid parameters to run a specified Quark script. This will run the script once on the provided calldata. Additionally, a `QuarkWallet` can execute `QuarkOperations` that have been signed by the wallet's `signer`. A `QuarkOperation` is defined by the following data fields:

- `nonce`: Identifier of the QuarkOperation
- `isReplayable`: Flag that is set to true if the script is intended to be executed more than once (replayable)
- `scriptAddress`: Address of the target script to execute
- `scriptSources`: Creation codes for script deployment
- `scriptCalldata`: Calldata for the target script containing the encoded function selector and arguments
- `expiry`: Expiration of operation signature that needs to be checked during signature verification

For executing such a `QuarkOperation` the user has to provide the structured Quark operation data as defined above, a valid signature of the Quark operation digest (produced by the wallet's `signer`) and a `submissionToken` that is tied to the operation `nonce`. Validating the `submissionToken` before script execution should mitigate indefinite script replays, and works as described in the subsection on "Replayability and Hash-Chaining".

2.2.1.1 Use of Transient Storage

The `QuarkWallet` uses transient storage to manage data relevant to script execution, including the active script address, active nonce, active submission token, and callback address. These values are stored in specific transient slots defined by the `QuarkWalletMetadata` library. For script execution, the Quark wallet uses the low-level `callcode` instruction to call the target script address, passing the provided calldata. Managing transient storage involves caching the previous values and setting the active ones before calling the external script. Once the execution flow returns, the transient storage is restored to the cached values.

2.2.1.2 Replayability and Hash-Chaining

The validity of a `submissionToken` is checked inside the `QuarkNonceManager.submit()` function. To prevent a replayable operation from being executed more often than intended, the number of replays is implicitly defined through the `submissionToken`. The user pre-computes a hash-chain, starting from a random 32-byte secret value s. For N+1 intended replays, the hash-chain looks as follows:

```
nonce = h^N(s) <- h^(N-1)(s) <- … <- h(h(s)) <- h(s) <- s
```

Starting from the `nonce`, each value along the chain represents a valid `submissionToken` for that nonce.

- For non-replayable scripts or the first replay of a replayable scripts, the following must hold: submissionToken == nonce.
- Subsequent executions of the same operation are only allowed if `isReplayable` = `true`. In such cases, the new `submissionToken` must be equal to the pre-image of the old `submissionToken`. In simpler terms: `hash(token_NEW)` == `token_OLD` must hold.

The hash function hash() is assumed to be pre-image resistant, meaning it is computationally infeasible to determine the pre-image of a given hash. Consequently, users must choose the intended number of replays N in advance to pre-compute the entire hash-chain.

This security mechanism ensures that an unauthorized party attempting to replay a valid signature has a negligible chance of guessing a valid subsequent `submissionToken`, effectively preventing replay attacks.

2.2.2 QuarkNonceManager

The `QuarkNonceManager` ensures that a wallet can execute an operation or script only if the corresponding (`nonce`, `isReplayable`, `submissionToken`) triple has been successfully validated and submitted.

It stores a `submissions` mapping that tracks the latest `submissionToken` for each `nonce` and associated wallet. A `nonce` represents a specific `QuarkOperation`, which may be replayable.

For a given `nonce`, a `submissionToken` is considered valid if it matches the next value in the hash chain (the pre-image of the previously submitted token) or the root value of the chain (for the first execution). This mechanism helps prevent unintended replays, as the correct `submissionToken` is computationally *hard* to compute without knowing the secret initial value s of the chain.

To mark an operation as non-replayable, a wallet can call the `QuarkNonceManager.cancel(nonce)` function, which will exhaust the given nonce and prevent further replays.

2.2.3 QuarkScript

The `QuarkScript` contract is an abstract contract and provides a framework for scripts within the Quark system. In addition to implementing helper functions, the key functionalities provided by `QuarkScript` include:

- Implementation of reentrancy guards `nonReentrant` and `onlyWallet`
- Reading from and writing to storage while managing storage isolation
- Setting and clearing the `CALLBACK_SLOT`
- Tracking the replay count of the currently active script

2.2.4 Quark Scripts

Legend Labs provides specific Quark scripts for certain operations: `Cancel.sol`, `MorphoScripts.sol` and `RecurringSwap.sol`. Quark scripts are contracts that implement additional wallet functionalities and are executed via the `callcode` instruction from a `QuarkWallet`.

2.2.4.1 Cancel

The `Cancel` script allows users to cancel one or more `QuarkOperations` by marking their corresponding identifiers (`nonce`) as exhausted. Once canceled, no further replays of the respective `QuarkOperation` can be executed.

The `Cancel` script can either be executed directly by the `executor` of the `QuarkWallet` or can be triggered if there is a valid signed `QuarkOperation` for the `Cancel` script with the specified `nonce`.

A replayable `QuarkOperation` for a certain `nonce` can also be cancelled by running the `nop()` function of the `Cancel` script within a QuarkOperation with the same nonce and `isReplayable` set to false.

2.2.4.2 MorphoScripts

For integrating with Morpho Blue, MetaMorpho, and Morpho rewards, the Quark wallets can execute the `MorphoActions`, `MorphoVaultActions` and `MorphoRewardsActions` scripts. The key functionalities of the Morpho scripts are the following:

- `MorphoVaultActions`: Manages deposits and withdrawals of ERC20 tokens into and from a MetaMorpho vault. It implements a `deposit` function that approves and deposits a specified amount of an asset into the vault, and a `withdraw` function that allows for withdrawing a specified amount or trying to withdraw the whole balance by redeeming all shares.
- `MorphoActions`: Enables asset and collateral management in a Morpho Blue market. The script implements the `repayAndWithdrawCollateral` function to repay borrowed assets and withdraw collateral, and the `supplyCollateralAndBorrow` function to supply collateral and borrow assets.
- `MorphoRewardsActions`: Implements a `claim` function for individual reward claims and a `claimAll` function for claiming rewards from multiple Morpho Universal Rewards Distributors in a single transaction. For each call to `MorphoUniversalRewardsDistributor.claim()`, a valid Merkle proof must be provided.

The Morpho script contracts do not inherit from `QuarkScript`.

2.2.4.3 RecurringSwap

The `RecurringSwap` script extends the `QuarkScript` contract and enables a Quark wallet to execute token swaps at regular intervals. Swaps are performed based on the configuration data passed to the contract's `swap` function, which defines the Uniswap `SwapRouter` to use, the input/output tokens, the swap path, and slippage parameters. A swap can only be executed if at least `interval` time has passed since the previous swap. The value of `nextSwapTime` is stored in the wallet's storage, indexed by a storage slot determined by the hash of the respective swap configuration.

The script uses Uniswap's `SwapRouter02` and relies on Chainlink price feeds. Both, `exactOutput` (fixed output amount) and `exactInput` (fixed input amount) swaps are supported. The former allows swapping as little as possible of token1 for a specified `amountOut` of token2, while the latter allows swapping a fixed `amountIn` of token1 for as much as possible of token2. If the two tokens being swapped do not have a direct liquidity pool between them a swap might require multiple hops.

2.2.5 Roles and Trust Model

- The QuarkWallet `executor` and `signer` are trusted.
- The seed used as the base of the hash-chain for replayable operations is assumed to come from a true source of randomness, in particular it has no known keccak pre-image.
- Scripts and scripts' calldata: Users should carefully verify the scripts they are interacting with, as well as the calldata. External calls performed by scripts are untrusted unless stated otherwise. If users sign operations that include untrusted scripts or untrusted calldata, they might suffer severe consequences, e.g., wallets might be drained, or arbitrary calls can be triggered from the context of wallets.
- Replayable transactions: some replayable transactions could be replayed by anyone during bad market condition, griefing the wallet. Users must make sure their replayable transactions implement checks in the scripts to limit the conditions under which the `QuarkOperation` can be reused.
- The `signer` is responsible to sign only trusted messages.

- Users submitting an externally signed `QuarkOperation` through `executeQuarkOperation()` and similar are untrusted. The quark operation itself is trusted, but it is assumed that it could be executed in a malicious context.
- Chainlink oracles used in `RecurringSwap` are assumed to update at least once every 24 hours. Timespans longer than 24 hours without updates are assumed to mean that the oracle is stale.

2.2.6 New in Version 2

In Version 2, nested execution is only allowed if initiated by the wallet itself. `callcode` has been replaced by `delegatecall`, and the `onlyWallet` modifier of `QuarkScript` has been dropped in favor of `nonReentrant`.

In the `RecurringSwap` script, the interval logic has been modified such that only one swap can be executed for every regularly spaced time window of a set length, placed at predefined intervals from the surrounding ones. This prevents an issue where if intervals were skipped, multiple trades could happen without waiting.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	1

- Reentrancy in Uniswap Flashloan Scripts Allows Arbitrary Execution Code Corrected

Medium-Severity Findings	4

- Callback Slot Is Not Reset Before Nested Operation Is Executed Code Corrected
- Nested Operations Can Invalidate Wallet State Code Corrected
- Price Oracle Data Not Checked for Staleness Code Corrected
- RecurringSwap Oracle Deviations Contributing to Slippage Specification Changed

Low-Severity Findings	1

- RecurringSwap Can Happen More Often Than Configured Interval Code Corrected

Informational Findings	5

- IQuarkWallet Interface Is Outdated Code Corrected
- Improvements for Event Emissions Code Corrected
- Inaccurate and Incomplete NatSpec Code Corrected
- Unreachable Code Code Corrected
- Unused Imports and Errors Code Corrected

6.1 Reentrancy in Uniswap Flashloan Scripts Allows Arbitrary Execution

Security High Version 1 Code Corrected

CS-LGN-QUARK2-001

Scripts `UniswapFlashLoan` and `UniswapFlashSwapExactOut` allow a wallet to receive a flashloan or a flashswap from Uniswap, and perform an arbitrary operation through `delegatecall` during the callback of the uniswap flashloan.

The script exposes two external functions: `run()` and `uniswapV3FlashCallback()`. The callback happens through the `uniswapV3FlashCallback()` function, after the script has enabled callbacks on the wallet by using `allowCallback()` in the `run()` function, which is called through `callcode` by the Wallet QuarkOperation execution.

In `uniswapV3FlashCallback()`, a `delegatecall` is performed toward an arbitrary address chosen by the flashloan initiator, which is supposed to be the Wallet owner. The target of the `delegatecall` is therefore fully trusted, however it could perform external calls (`CALL` opcode) to external third parties, which are possibly untrusted (interacting with an upgradeable contract for example).

While the `delegatecall` in `uniswapV3FlashCallback()` is executing, the Wallet can potentially be re-entered through the `run()` function, through the QuarkWallet `fallback()` function. This allows performing a second flashloan, which results in a `delegatecall` to an address specified by the flashloan initiator. If the initiator of the re-entrant call to `run()` is an untrusted third party, they can run arbitrary code on the wallet and compromise all owner's funds.

Code partially corrected:

The code has been updated in the `quark` and `quark-script` repos to use `disallowCallback()` immediately after `uniswapV3FlashCallback()` is entered.

6.2 Callback Slot Is Not Reset Before Nested Operation Is Executed

Correctness | Medium | Version 1 | Code Corrected

CS-LGN-QUARK2-002

During nested operation execution, in `executeScriptInternal`, the callback slot is not reset before calling into the provided script of the nested operation via `callcode`. This could introduce a vulnerability as the outer operation callback may still be valid during a nested quark operation, potentially leading to unintended execution flows.

For example, problems may arise if the inner operation was signed with the assumption that the fallback method would revert, but the outer callback interferes, leading to unexpected behaviors.

Code corrected:

In `executeScriptInternal`, a `tstore(callbackSlot, 0)` instruction has been added before the `delegatecall` to the script, which clears the callback for the wallet. Now, in case of nested execution, the inner script does not inherit the outer script's callback.

6.3 Nested Operations Can Invalidate Wallet State

Design | Medium | Version 1 | Code Corrected

CS-LGN-QUARK2-003

Currently, nested quark operations are allowed. More specifically, a wallet may execute an operation that executes another (signed) operation. Any level of nesting is allowed. Once the innermost script completes, execution is passed on to the script that invoked it, continuing until the outermost operation terminates. Nested operations are thus completed in a LIFO-like order. For scripts that rely on the state of the wallet, this can lead to unwanted behavior. Consider the following example:

1. Assume that `WETH.balanceOf(wallet) = x, (x > 0)`.
2. The wallet executes operation A which relies on the current value of `WETH.balanceOf(wallet)`.
3. Operation A executes operation B which changes the wallet's WETH balance.
4. Execution returns to operation A which now operates on a changed state, possibly invalidating invariant assumptions as `WETH.balanceOf(wallet) != x`.

Thus, the possibility to interleave quark operations may cause incorrect behavior. This is especially sensitive when the nested operation is not executed by the wallet directly, but by a third party system that has been called by the wallet through an external call. In that case, the wallet does not expect that a nested operation has been executed when the external call returns.

Code corrected:

The following condition has been added in `executeScriptInternal`:

```
if and(iszero(eq(oldActiveScript, 0)), iszero(eq(caller(), address()))) {
    ...
    revert(add(ptr, 0x1c), 0x04)
}
```

Only a self-call from the wallet can now trigger a nested operation execution. If an external address tries to execute an operation on a wallet that is already executing another operation, the call will fail. This prevents the wallet from unknowingly operating on invalid state caused by an unexpected nested execution.

Wallet users and script authors should still be careful when crafting operations that contain nested operations, to avoid invalidating the wallet's state.

6.4 Price Oracle Data Not Checked for Staleness

Correctness Medium Version 1 Code Corrected

CS-LGN-QUARK2-004

`RecurringSwap` computes the expected in/out token amount using oracles conforming to the Chainlink interface `AggregatorV3Interface`. Chainlink oracles could be stale, if they have not been updated recently enough, especially in cases of network congestion, or sequencer downtime on L2s. The `RecurringSwap` script does not check oracle staleness, possibly resulting in more slippage than expected.

Code corrected:

Legend Labs mitigates the issue by verifying that the price has been updated within the last 24 hours.

6.5 RecurringSwap Oracle Deviations Contributing to Slippage

Design Medium Version 1 Specification Changed

CS-LGN-QUARK2-005

`RecurringSwap` relies on Chainlink oracles to provide a token amount upon which to calculate tolerable slippage. This token amount is computed by potentially aggregating multiple ChainLink oracles. Since Chainlink prices are always quoted in ETH or USD, in the worst case there might be 3 oracles used (token1 to USD, USD to ETH, ETH to token2). Each oracle has a deviation threshold, a percentage value that the spot price must deviate from the current oracle price in order to trigger an on-chain update. This means that the three oracles used can be outdated up to their deviation threshold. As the errors caused by the deviation thresholds accumulate when combining the three oracles, the token amount used to compute the slippage could significantly differ from the value computed with up-to-date spot prices, meaning more slippage can be incurred.

The deviation threshold of general Chainlink feeds is around 2% for arbitrary tokens, while the USD/ETH feed has a 0.5% deviation threshold. Combining two crypto feeds and the USD/ETH feed results in possibly 4.5% more slippage than expected.

Comment:

Legend Labs accepts the behavior as inherent to the system. Users are made aware that they can incur more slippage than expected by the following note added in the `RecurringSwap` script documentation in Version 3:

```
/// Note: The deviation threshold of general Chainlink feeds is between 0.5-2%.
///       It's important to consider that using multiple price feeds can amplify
///       the potential for slippage beyond the expected limits.
```

6.6 RecurringSwap Can Happen More Often Than Configured Interval

Design Low Version 1 Code Corrected

CS-LGN-QUARK2-007

In `RecurringSwap`, the next swap time is stored when performing a swap by increasing the currently stored `nextSwapTime` by `interval`. If the stored `nextSwapTime` is further in the past than `interval`, another swap can immediately be executed after the current one. This could cause swaps to be executed at the same time, instead of waiting `interval` between swaps, in case some swaps have been skipped because of offline QuarkOperation executors, or network congestion. This could in turn result in more MEV extraction and less averaging of purchase prices in the swaps.

Code corrected:

The code has been corrected and now uses `SwapWindows`, which have a window `length` and an `interval` duration. Windows start at fixed intervals, and the code now ensures that at most one swap per window may be executed. `SwapWindows` remain open until `length` seconds after the start time of the respective window. The choice of parameters distinguishes how closely swaps can occur, ensuring that swaps are spaced by at least $interval - length$ seconds, when $length < interval$.

6.7 IQuarkWallet Interface Is Outdated

Informational Version 1 Code Corrected

CS-LGN-QUARK2-008

IQuarkWallet, defined in `quark-core/src/interfaces/IQuarkWallet.sol`, is outdated.

Some methods are missing (e.g. `executeQuarkOperationWithSubmissionToken()`). Some methods are defined but no longer exist in QuarkWallet: `executeScriptWithNonceLock()`.

Moreover, `QuarkWallet` does not implement `IQuarkWallet`.

Code correct:

The `IQuarkWallet` interface has been updated accordingly.

6.8 Improvements for Event Emissions

Informational Version 1 Code Corrected

CS-LGN-QUARK2-009

1. The `QuarkExecution` event misses a data field recording the `signer` of the operation executed. This can be used to aggregate events from different wallets with the same signer, or in cases where the wallet's `signer` can change.
2. The `NonceSubmitted` event does not index the address of the wallet. This could be useful for filtering events.
3. The `NonceSubmitted` event is emitted upon successfully submitting a nonce. However, for non-replayable nonces the `submissionToken` is passed as event argument instead of the `EXHAUSTED` value, despite the `submissions` mapping being correctly set to `EXHAUSTED`.

Code corrected:

1. No change. Legend Labs states:

   > There are already three indexed fields on the event, each of which we think are more important to index on than the wallet's signer address. This is primarily because the signer address is immutable, so one could always figure out the signer address given the wallet address. The same cannot be said about the other indexed fields (executor, scriptAddress, nonce).

2. Code changed. The event now indexes the wallet and the nonce.
3. Code changed. For non-replayable nonces, the EXHAUSTED value is used in the event.

6.9 Inaccurate and Incomplete NatSpec

Informational Version 1 Code Corrected

CS-LGN-QUARK2-010

The NatSpec is incomplete or potentially inaccurate in some cases. Below is a (potentially incomplete) list of examples:

1. The NatSpec of the constant definitions in `QuarkWallet.sol` for the transient storage slots (`CALLBACK_SLOT`, `ACTIVE_SCRIPT_SLOT`, `ACTIVE_NONCE_SLOT`, `ACTIVE_SUBMISSION_TOKEN_SLOT`) refers to "storage" instead of "transient storage".
2. The `REENTRANCY_FLAG_SLOT` NatSpec in the `QuarkScript` contract mentions "storage" instead of "transient storage".
3. The NatSpec of the `submit()` function of the `QuarkNonceManager` is misleading and inaccurate:

   a) The documentation for the `nonce` parameter is potentially misleading due to the missing context in which "chain" is used.

   b) For the `submissionToken` parameter it mentions: "For single-use operations, set *submissionToken* to *uint256(-1)*. For first-use replayable operations, set *submissionToken* = *nonce*. Otherwise, the next submission token from the nonce-chain."

   This is incorrect. For single-use operations, the `submissionToken` must be equal to the `nonce`, and `isReplayable` must be set to false.

4. The NatSpec of the `nonReentrant` modifier in `QuarkScript` mentions:

"A safer, but gassier reentrancy guard that writes the flag to the QuarkNonceManager"

This is incorrect, as the state is stored in the wallet's transient storage.

5. The `QuarkScript` contract lacks NatSpec for most functions.
6. The `RecurringSwap` contract includes the following `@dev Note` about the storage layout:

```
/**
* @dev Note: This script uses the following storage layout in the QuarkNonceManager:
*         mapping(bytes32 hashedSwapConfig => uint256 nextSwapTime)
*             where hashedSwapConfig = keccak256(SwapConfig)
*/
```

The storage layout displayed does not represent the one used in the code, and mentioning the `QuarkNonceManager` is incorrect as the storage is in the wallet itself.

Code correct:

The NatSpec has been updated accordingly.

6.10 Unreachable Code

Informational **Version 1** **Code Corrected**

CS-LGN-QUARK2-014

In `QuarkScript.getActiveReplayCount()`, the following condition is checked:

```
if (submissionToken == QuarkNonceManagerMetadata.EXHAUSTED) {
    return 0;
}
```

`submissionToken` will never be equal to `EXHAUSTED` in this context, as this is the value provided by the user of the wallet and submitted to the nonce manager, which asserts that it is different from `EXHAUSTED`.

Code corrected:

The unreachable code has been removed.

6.11 Unused Imports and Errors

Informational **Version 1** **Code Corrected**

CS-LGN-QUARK2-015

1. The `IQuarkWallet` interface is imported in `QuarkNonceManager.sol` and `QuarkWallet.sol` but is unused in both files.
2. The `QuarkWallet` contract is imported in `QuarkScript.sol` and `RecurringSwap.sol`, however it is unused.
3. Errors `InvalidActiveNonce` and `InvalidActiveSubmissionToken` are defined in `QuarkScript.sol`, but they are never used.

Code corrected:

1. changed accordingly
2. `RecurringSwap.sol` still imports `QuarkWallet`
3. changed accordingly

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Misleading Nomenclature

Informational **Version 1** Acknowledged

CS-LGN-QUARK2-011

The identifier for a `QuarkOperation` is referred to as `nonce` in the codebase. For replayable Quark operations, the same operation under the same signature, same `nonce` and new `submissionToken` can be executed multiple times. This makes the use of the variable name `nonce` (number used once) misleading. This inconsistency may impact the clarity of both the codebase and its documentation.

Comment:

Legend Labs states:

> The *nonce* in question has to be unique for each Quark Operation, functioning as a nonce (a number used once) at the Quark Operation level. While the same replayable Quark Operation can be submitted multiple times with a valid submission token, the use of the same nonce for different operations is not permitted.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Number of QuarkOperation Replays Implicit

Note **Version 1**

A replayable `QuarkOperation` with a `nonce` derived from hashing the random secret s `N` times is replayable up to `N+1` times and not more, only if the following assumptions hold:

- The secret s is chosen uniformly at random. If an attacker can predict s, they could pre-compute the hash-chain and trigger script replays before the user does.
- The random secret s must not have a known pre-image.
- The pre-computed hash-chain must not contain a hash that equals `0x000...0` or `0xfff...f` as this would prevent any further replays.
- `submissionTokens` (hashes in the hash-chain) must not overlap for any wallet address, or chain (network). If they do, an attacker could reuse a previously observed pre-image to trigger a replay.

If any of the above assumptions are violated, an attacker may successfully submit a valid pre-image to the `QuarkNonceManager` and no guarantees on the number of `QuarkOperation` replays can be made.

8.2 Possibly Unbounded Gas Consumption for `getActiveReplayCount()`

Note **Version 1**

The for-loop in `QuarkScript.getActiveReplayCount()` iteratively hashes the active submission token of the active nonce. The loop terminates when a hash is produced that equals the active `nonce`. The number of submission tokens and therefore the number of intended replays N is implicitly defined through the hash-chain. For large N, gas consumption will grow and could be very high.

We refer to *intended* replays due to the u.a.r. property assumptions of the secret s, mentioned in the Note Number of QuarkOperation replays implicit.

8.3 Wallet Callback Can Be Called Multiple Times

Note **Version 1**

CS-LGN-QUARK2-006

The `fallback` function of the `QuarkWallet` makes a `delegatecall` to the callback address. Since the callback remains set in the wallet's transient storage, multiple calls to the callback are possible. Within a quark operation where the script performs an external call into an untrusted contract, this may lead to unintended code execution and possible vulnerabilities, as the wallet can be reentered.

Developers of scripts must be well aware that the callback remains enabled after being used. As the security of the wallet depends on the security of the scripts, developers should manually disable the callback using the `disallowCallback()` function after the callback has been executed and before performing any external calls to protect the wallet from unexpected reentrancies.