PUBLIC

Code Assessment

of the Quark Smart Contracts

June 12, 2024

Produced for

by

Contents

1 Executive Summary

Dear Compound Team,

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

Compound implements Quark Wallets which is a system for account abstraction based on wallet contracts that can run arbitrary code (scripts), deployed by a special contract `CodeJar`. Users can then trigger actions from their wallets by executing scripts directly or signing messages according to EIP-712 format.

The most critical subjects covered in our audit are access control, signature handling, functional correctness, gas griefing and front-running. Security regarding all the aforementioned subjects is satisfactory.

The general subjects covered are code complexity, trustworthiness, documentation and gas efficiency. The codebase is generally well written and includes inline comments that improve the readability of code. Contracts in scope are not upgradable and do no have privileged roles, hence providing a high level of trustworthiness.

The system offers flexibility and new features can be plugged in by scripts. We would like to emphasize that developers should carefully assess new scripts to avoid introducing vulnerabilities that can exploit user's wallets. Users should also carefully evaluate scripts and their parameters. Interacting with a malicious script or passing wrong parameters to a verified script could be enough to exploit a wallet.

In summary, we find that the codebase provides a satisfactory 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	4
Low-Severity Findings	7
• Code Corrected	4
• Risk Accepted	1
• Acknowledged	2

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 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	28 Nov 2023	0b49126788b33e0b014bd08ccf972f3b3b39d8bb	Initial Version
2	17 Jan 2024	d67860fa8fc4e94cf54d3ee6931ca52c5d4ea07d	Version with fixes
3	05 Feb 2024	e8e4e37327aa2b9e5119327199f53de29479755b	Final Version
4	21 May 2024	e1da89dbc792bf1f36a1bddd0a8ceadb2a4b85db	Version with new scripts
5	10 Jun 2024	c7a8ff588056d41ec20509f56fe03f351306a49a	Fixes for the new scripts

For the solidity smart contracts, the compiler version `0.8.19` and evm version `paris` were chosen. After Version 2 the compiler version was updated to `0.8.23`.

The following contracts are in the scope of the review:

```
CodeJar.sol
QuarkScript.sol
QuarkStateManager.sol
QuarkWallet.sol
QuarkWalletFactory.sol
core_scripts:
    Ethcall.sol
    Multicall.sol
    UniswapFlashLoan.sol
    UniswapFlashSwapExactOut.sol
    lib:
        UniswapFactoryAddress.sol
periphery:
    BatchExecutor.sol
terminal_scripts:
    TerminalScript.sol
    interfaces:
        IComet.sol
        ICometRewards.sol
vendor:
    uniswap_v3_periphery:
        PoolAddress.sol
```

After Version 2, the codebase has been refactored. The new scope is:

```
codejar:
    src:
        CodeJar.sol
        CodeJarStub.sol
quark-core:
    src:
        QuarkScript.sol
        QuarkStateManager.sol
        QuarkWallet.sol
        periphery:
            BatchExecutor.sol
quark-core-scripts:
    src:
        Ethcall.sol
        Multicall.sol
        UniswapFlashLoan.sol
        UniswapFlashSwapExactOut.sol
        lib:
            UniswapFactoryAddress.sol
        vendor:
            uniswap_v3_periphery:
                PoolAddress.sol
quark-factory:
    src:
        QuarkFactory.sol
quark-proxy:
    src:
        QuarkMinimalProxy.sol
        QuarkWalletProxyFactory.sol
terminal-scripts:
    src:
        TerminalScript.sol
        interfaces:
            IComet.sol
            ICometRewards.sol
```

After Version 3, `terminal-scripts/` and `TerminalScript.sol` have been renamed `legend-scripts/` and `LegendScript.sol`.

After Version 4, some contracts have been added/deleted/moved (A/D/M) to/from/within the scope. The changes from the previous scope are highlighted below:

```
codejar:
    src:
        (A) CodeJarFactory.sol
        (D) CodeJarStub.sol
quark-core:
    src:
        (A) QuarkWalletStandalone.sol
        (A) interfaces:
            (A) IHasSignerExecutor.sol
            (A) IQuarkWallet.sol
quark-core-scripts:
    src:
```

```
        (A) Paycall.sol
        (A) Quotecall.sol
        vendor/chainlink:
            (A) AggregatorV3Interface.sol
(D) legend-scripts:
    (D) src:
        (M) LegendScript.sol (moved to test/lib/DefiScripts.sol)
        (D) interfaces:
            (D) IComet.sol
            (D) ICometRewards.sol
```

2.1.1 Excluded from scope

The following contracts are explicitly not in the scope of the review:

```
core_scripts:
    ConditionalMulticall.sol
    lib:
        ConditionalChecker.sol
```

The out-of-scope contracts after Version 2 are:

```
quark-core-scripts:
    src:
        ConditionalMulticall.sol
        lib:
            ConditionalChecker.sol
```

Additionally, any contracts or scripts that might be deployed in the future but are not explicitly listed above are out of the scope of this review. Third-party libraries are out of the scope of this review. Finally, other Compound components like Comet were not in scope of this review and are assumed to always behave non-maliciously, correctly, and according to their specification. The price feeds for the `Paycall` and `Quotecall` scripts are out of the scope of this review.

2.2 System Overview

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

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

Compound offers account abstraction in the form of a wallet system that can run arbitrary code (scripts), deployed by a special `CodeJar` contract. Users can then trigger actions from their wallets by signing messages with EIP-712.

2.2.1 CodeJar

The `CodeJar` contract allows the deployment of so-called scripts. A script can be arbitrary bytecode with the limitation that the constructor and storage variable initialization upon deployment are not supported. The scripts are deployed with the `CREATE2` opcode with salt `0x00`.

2.2.2 QuarkWallet

A wallet has a signer and an executor. The actions the wallet can execute are called operations. The signer is the owner of the wallet and signs the operations that must be executed with EIP-712. The executor has a privileged role and can submit operations without a signature to the wallet. The signer can be an EOA or a smart contract. If the signer is a smart contract, it must implement the EIP-1271. The state of the wallet should not be assumed to be held in the wallet's storage, it is managed by the `QuarkStateManager`. A wallet can trigger `selfdestruct` through a script which deletes its code and state from the address and transfers out its Ether balance, however, the wallet can be redeployed with the same salt from the factory. The flow of operations in the wallet is described below.

2.2.3 Flow of QuarkOperations

An operation consists of a target script address, some calldata, and a nonce. During an operation, the target script can read and write the currently active storage context in the `QuarkStateManager`. A nonce is associated with at most one script address and can be reused if cleared during an operation. When an operation is submitted to a wallet, it will call `QuarkStateManager.setActiveNonceAndCallback()` to check and set the nonce and the script address. The `QuarkStateManager` triggers a callback in the wallet (`executeScriptWithNonceLock()`), where the target script address is called with a low level `CALLCODE` along with the calldata. Quark operations have an expiry and they are considered valid only if `expiry` is in the future (`block.timestamp < op.expiry`).

Scripts can enable callbacks to the Wallet through the `fallback` function by setting a non-zero address at the key `CALLBACK_KEY`. The fallback function will then `DELEGATECALL` to this address only. Scripts can allow nonce replay by clearing the active nonce, i.e. only the nonce associated to the current execution context, by calling `QuarkStateManager.clearNonce()`. Note that scripts can set arbitrary nonces.

2.2.4 QuarkStateManager

The `QuarkStateManager` keeps different mappings related to the execution context of `QuarkOperations`:

- `nonces`: tracks the nonces per wallet. Once a nonce is set, it can only be cleared while active (see `activeNonceScript`). Clearing a nonce marks it for reuse.
- `nonceScriptAddress`: stores the script address associated to a nonce that was cleared to avoid nonce replay with different script address.
- `activeNonceScript`: holds the currently active nonce/script address pair for a wallet, outside of an execution context the pair is `(0, address(0))` for every caller. There is always at most one active pair per wallet. In the case of nested `QuarkOperation` for a wallet, the pairs form a stack, the active pair being on top of the stack.
- `walletStorage`: key-value mapping indexed by wallet address and nonce. Can be used as storage for the execution context of a nonce, the storage can be accessed (read/write) only for the `msg.sender` and currently active nonce.

The main function of this contract is `setActiveNonceAndCallback`. The function is assumed to be called from a wallet, it first checks that the nonce to be used is not set and reverts if it's the case. Then the nonce is set in the `nonces` mapping for the calling wallet and the script address is checked to be the same as the address associated with the nonce if it was marked for reuse. The currently active nonce is pushed back on the stack and the new nonce/script address pair is pushed on the stack by being set as the `activeNonceScript` for the calling wallet. Next, the calling wallet is called back with `Wallet.executeScriptWithNonceLock()`, when the callback returns, the script address is recorded in the `nonceScriptAddress` if the nonce was cleared during the callback. Lastly, the stack of active nonce script is popped to fully exit the execution context of the nonce/script address pair. Note that

in the special case where `scriptA` with nonce 42, clears it and reenters the wallet with `scriptB` and the same nonce, and `scriptB` does not clear it, the nonce will not be replayable.

The contract implements a helper function `nextNonce` which returns the next nonce that can be used for a wallet. This function implements a `for-loop` that might run out-of-gas, hence should be called mainly off-chain.

2.2.5 QuarkWalletFactory

The factory deploys the wallets with the `CREATE2` opcode and an arbitrary salt. The deployment of wallet is permissionless and wallets can be deployed for arbitrary signers and the default salt is `0x00`. If the salt is `0x00`, the deployed wallet has no executor (`address(0)`), but this wallet will be the executor of all the other wallets deployed for the same signer with a non-zero salt.

2.2.6 Scripts

Scripts are arbitrary contracts that act in the context of a `Wallet` via `callcode`. They provide functionalities that are not native in the `Wallet`. Scripts can always overwrite their own nonce-scope storage in the `QuarkStatemanager`, including their callback key and their reentrancy lock.

2.2.7 Changes in Version 2:

- The codebase has been refactored and split into different submodules.
- The wallets are deployed behind a minimal proxy by the `QuarkWalletProxyFactory`.
- The system is deployed and linked together by the `QuarkFactory`.
- The `executor` of a `Wallet` can be any arbitrary account, so both `executor` and `signer` of a wallet can potentially be the same account.
- The functions `QuarkWallet.executeQuarkOperation` and `QuarkWallet.executeScript` are no more payable, meaning that `ETH` must either be sent upfront, or pulled during a script execution.
- In `QuarkWallet.executeScriptWithNonceLock`, the `msg.value` is set to `0` for the calling context of `callcode`.
- A new modifier `onlyWallet` has been added. It acts as a weaker form of reentrancy protection as it would only allow the `Wallet` to reenter itself through a script that is assumed to be trusted.
- The function `QuarkStateManager.setActiveNonceAndCallback()` is now caching the `scriptAddress` before making the callback to the `Wallet`. This slightly changes the behavior of the case where the execution of `scriptA` with nonce 42 is reentered with `scriptB` and the same nonce and both scripts clear the nonce, the code in **Version 1** will set the `nonceScriptAddress` for that nonce to `scriptB`, and the code in **Version 2** will set it to `scriptA`.

2.2.8 Changes in Version 4:

- The `CodeJar` has been updated to directly deploy the provided deployment bytecode, and not some fixed constructor code appended to the runtime bytecode anymore.
- A new way to submit operations was added: execute multi Quark operation. This allows a signer to sign a batch of operation's hashes that do not necessarily target the same chain of a wallet, instead of signing each operation individually. The batch can then be submitted to the wallet, along one of the operation of the batch that will be executed.
- Two new scripts have been added: `Paycall` and `Quotecall`. Both allow to pay the `tx.origin` in some ERC20 tokens for submitting the transaction. In `Paycall`, the gas used to make the call is recorded and `tx.origin` receives the equivalent gas value in the payment token after the call, up

to `maxPaymentCost`. In `Quotecall`, the gas used to make the call is recorded and `tx.origin` receives some token amount (`quotedAmount`) before the call, if the gas used is not within `maxDeltaPercentage` of `(actualAmount - quotedAmount) / quotedAmount`, the transaction reverts.

2.2.9 Changes in Version 5:

- In the scripts `Paycall` and `Quotecall`, that allow users to pay for gas in ERC20, the `propagateReverts` variable has been removed. This implies that if the inner call that a user wants to execute reverts, the whole transaction will revert. This was done to avoid potential gas griefing at the expense of the users (see Gas griefing attacks).
- The `GAS_OVERHEAD` constant, previously set to 67 500, was changed to 135 000 in `Paycall` and to 100 000 in `Quotecall` (see Inconsistent GAS_OVERHEAD Value).

2.2.10 Trust Model

- Scripts and scripts' calldata: users should carefully verify the scripts they are interacting with, as well as the calldata. Typically, if a script implements a proxy, users may want to monitor the implementation changes. 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 `Quark` operation can be reused.
- Users: not trusted. However, users are responsible to sign only messages that they trust.
- `Paycall` and `Quotecall` scripts: the price feeds connected to these scripts are expected to be carefully chosen.

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	3

5.1 Contracts Do Not Extend Their Interfaces

Design Low Version 4 Acknowledged

CS-QUARK-017

Interfaces are likely to be used by integrators and thus should represent the functions and structs of the contract. To ensure this, the contracts should extend their interfaces. Here is a non-exhaustive list of contracts that do not extend their interfaces:

- `QuarkWallet` should extend `IQuarkWallet`
- `QuarkMinimalProxy` should extend `IHasSignerExecutor`

Acknowledged:

In Version 5, `QuarkMinimalProxy` has been updated to extend `IHasSignerExecutor`.

On the other side `QuarkMinimalProxy` stays unchanged. Compound had declared that this behavior is intended:

```
We purposely did not extend QuarkWallet with IQuarkWallet because we prefer the QuarkOperation
struct to be defined directly in the QuarkWallet implementation for better readability.
```

5.2 A Nonce Can Be Used With Different Target Addresses

Design | Low | Version 1 | Acknowledged

CS-QUARK-004

If during a script operation with nonce `X` and target address `A`, the nonce is cleared and the wallet is reentered with another script operation with nonce `X` and target `B`, the nonce `X` has been used for addresses `A` and `B`. This is possible because the target address is recorded only after the callback to the `QuarkWallet`. This breaks the assumption that a nonce can be associated with at most once target script address.

Acknowledged:

Compound considers this behavior in line with specifications and provides the following reasoning:

```
We posit that the invariant always holds true post-execution of a script. This means
that after a Quark script has fully executed, there should be no way for a nonce to
be used with a different script address. We explored keeping this invariant true in
all cases (even before the script has finished fully executing) by writing the script
address before executing a script, but this introduces a 15k+ gas overhead and we
decided it was not worth it.
```

5.3 Gas Griefing Through Memory Return Bomb

Design | Low | Version 1 | Risk Accepted

CS-QUARK-006

The functions using the pattern `(bool success, bytes memory data) = callX` can be the source of gas griefing if the target contract returns a memory pointer that triggers unnecessary memory extension (return bomb).

Risk accepted:

Compound acknowledged the issue and decided to keep the code unchanged giving the following reasoning:

```
Since the Quark wallet already gives the callee all the gas, the callee can gas
grief through other means. We don't believe this to be an issue that Quark should
address because users are ultimately responsible for the transactions they sign.
```

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

- Gas Griefing Attacks Code Corrected

Medium-Severity Findings	4

- Inconsistent GAS_OVERHEAD Value Code Corrected
- Possible Reentrancy in ERC20 Transfer Code Corrected
- Script Addresses Without Code Are Undetected Code Corrected
- Scripts Deployment Can Be Blocked Code Corrected

Low-Severity Findings	4

- Consistency of Empty Code Existence Code Corrected
- Gas Griefing Through Script Implementation Destruction Code Corrected
- Limited Events Emitted by Contracts Code Corrected
- Superfluous Allowance Given in supplyFrom Code Corrected

Informational Findings	5

- Gas Griefing Through BatchExecutor Failure Code Corrected
- Unused Code Code Corrected
- Unchecked Loop Increment Code Corrected
- Possible Allowance Leftover in UniswapSwapActions Code Corrected
- QuarkScript Can Be Marked as Abstract Code Corrected

6.1 Gas Griefing Attacks

Design High Version 4 Code Corrected

CS-QUARK-013

The new scripts `Paycall` and `Quotecall` open the way for griefing attacks, assuming `GAS_OVERHEAD` covers the gas cost of the transaction outside of `initialGas - gasleft()`. Ignoring any arbitrage opportunity offered by the Quark operation, this assumption holds as no one would execute a pay/quotecall otherwise, as they would be losing money in the transaction. The signer signs the operation and makes it public, the submitter picks the signature and submits it to the `QuarkWallet`.

There are two distinct scenarios where a griefing attack can happen:

- Case `propagateReverts` is true: The signer can include some calldata that burns a lot of gas and make the call revert, forcing the submitter to pay a lot of gas for a reverting transaction, without the submitter (`tx.origin`) receiving payment for the gas. Even if the submitter simulates the

transaction before submitting it, an observer can frontrun it and update some on-chain state that will make the transaction fail.

- Case `propagateReverts` is false: The submitter can submit the transaction when they are guaranteed that it will fail, making the signer pay for a failing transaction on purpose, provided the wallet has enough of the payment token. In the case of `Paycall`, the value extracted from the signer for one failed operation over one transaction is `EV = min(maxPaymentCost, (GAS_OVERHEAD + (PaycallOperationGasCost)) * gasPriceInToken)`. The submitter pays `gasTX` to send the operation on-chain. In the normal use case where the operation succeeds, we expect `EV - gasTX > 0` to be a fee taken by the submitter for the service. In the adversarial case where the submitter knows for sure that the operation will fail, this amount becomes theft, as the signer may need to sign the same operation and pay for gas again to reach their goal. It is possible for the submitter to increase their margin by batching multiple operations in one transaction.

Code corrected:

In Version 5, the `propagateReverts` variable was removed and consequently reverts are always propagated; if the inner call reverts, the whole transaction will revert. This effectively prevents the second attack described, where griefing could be done at the expense of the user.

Therefore, the scenarios above change as follow:

- Case `propagateReverts` is true: Compound acknowledged the potential risks and explained their reasoning as follows:

```
Submitters are responsible for simulating operations before submission to make sure they are paid appropriately.
In the case that a griefer constructs a call that reverts non-deterministically, submitters could either submit
the transaction through Flashbots to guarantee no reverts or choose to only submit calls to known/whitelisted contracts.
```

- Case `propagateReverts` is false: the issue is not present anymore.

6.2 Inconsistent GAS_OVERHEAD Value

Correctness | Medium | Version 4 | Code Corrected

CS-QUARK-014

Both `Paycall` and `Quotecall` have a `GAS_OVERHEAD = 67_500`. In `Paycall`, this value includes the gas for the transfer of the payment token, but in `Quotecall` the gas for token transfer is also included in `gasInitial - gasleft()`. In summary, `GAS_OVERHEAD` has the same value but does not cover gas for the same steps, one of them should be lower/higher.

Code corrected:

Compound has corrected the code in Version 5: `GAS_OVERHEAD` was set to 135k for `Paycall` and to 100k for `Quotecall`. The amounts were derived by summing the initial gas required by the transaction (21k + calldata gas) with an estimation of the gas used by the Quark Operation wrapping the inner call (70k). For `Paycall`, 35k of gas was added to account for the ERC20 transfer.

6.3 Possible Reentrancy in ERC20 Transfer

Security | Medium | Version 1 | Code Corrected

CS-QUARK-001

The function `transferERC20Token` in `TerminalScript` opens a reentrancy possibility (similar to `transferNativeToken()`) for tokens that implement transfer hooks such as ERC-777 tokens. The function does not use the `nonReentrant` lock, hence making the script vulnerable to reentrancy for specific tokens.

Code corrected:

Both functions `transferERC20Token()` and `transferNativeToken()` are now protected with the modifier `onlyWallet`. This modifier behaves differently from `nonReentrant` and is implemented as below:

```
modifier onlyWallet() {
    if (msg.sender != address(this)) {
        revert ReentrantCall();
    }
    _;
}
```

As stated in the natspec comment of the modifier, reentrancies initiated from the wallet itself are still possible:

```
@notice A cheaper, but weaker reentrancy guard that does not prevent recursive
 reentrancy (e.g. script calling itself)
```

6.4 Script Addresses Without Code Are Undetected

Design | Medium | Version 1 | Code Corrected

CS-QUARK-002

The functions `QuarkWallet.executeScript` and `QuarkWallet.executeQuarkOperation` assume some code resides at the `scriptAddress` and do not check the code length at this address. The transaction will not revert because performing `callcode()` on an empty account will always succeed.

For example, a transaction using an already deployed script can be front-run by calling `selfdestruct` on the target script implementation if it allows it. The transaction executing the quark operation will succeed but will have no effect other than consuming the nonce.

Code corrected:

The function `QuarkWallet.executeScriptWithNonceLock` has been revised to revert if the target `scriptAddress` has empty code:

```
if (scriptAddress.code.length == 0) {
    revert EmptyCode();
}
```

This function is triggered from both `executeScript()` and `executeQuarkOperation()`, hence transactions now revert instead of silently succeeding when the target `scriptAddress` is empty.

6.5 Scripts Deployment Can Be Blocked

Design | Medium | Version 1 | Code Corrected

CS-QUARK-003

The function `CodeJar.saveCode` assumes that if the target address does not contain code, its codehash will be equal to `0`:

```
if (codeAddressHash == 0) {
    // The code has not been deployed here (or it was deployed and destructed).
    ...
} else if (codeAddressHash == keccak256(code)) {
    // Code is already deployed and matches expected code
    ...
} else {
    // Code is already deployed but does not match expected code.
    // Note: this should never happen except if the initCode script
    //       has an unknown bug.
    revert CodeHashMismatch(codeAddress, keccak256(code), codeAddressHash);
}
```

However, the assumption that if there is no code in the address where a script will be deployed has a codehash of `0` is wrong. If Ether is sent to the target address, its codehash will change to `keccak256("")` which is the same as codehash of EOAs. This behavior can be abused by front-running the deployment of script by sending at least `1 wei` to the target address, forcing `saveCode()` into the reverting branch and making the script impossible to deploy from the `CodeJar`.

Furthermore, the attacker can also destroy existing scripts by triggering a call to `selfdestruct`. This is possible for scripts that perform `delegatecalls` into user-provided addresses, like `Multicall`. Once the script is destroyed, the attacker can send `1 wei` of Ether to the address such that its codehash becomes non-zero, hence `saveCode` will always revert in the future if someone wants to deploy the same script.

Code corrected:

The function `CodeJar.saveCode` has been revised to always deploy the new `code` if the existing codehash of the target address does not match the hash of the `code`:

```
if (codeAddressHash == keccak256(code)) {
        // Code is already deployed and matches expected code
        return codeAddress;
    } else {
        // The code has not been deployed here (or it was deployed and destructed).
    }
```

6.6 Consistency of Empty Code Existence

Correctness | Low | Version 1 | Code Corrected

CS-QUARK-005

The function `CodeJar.codeExists` returns `false` when the target address' code does not match the deployed bytecode. However, it is inconsistent in the particular case of empty bytecode. By default, `codeExists()` will return `false` when `code=""`, as the codehash of the derived address will be `0`. If

some ETH was to be sent to the derived address, its codehash will be equal to keccak256("") and codeExists() will return true even though the empty bytecode has not been "deployed".

Code corrected:

The function codeExists() now returns true only if the code length at the target address has a non-zero length:

```
return codeAddress.code.length != 0 && codeAddress.codehash == keccak256(code);
```

Note that calling the function with empty code codeExists(code="") returns always false.

6.7 Gas Griefing Through Script Implementation Destruction

Design | Low | Version 1 | Code Corrected

CS-QUARK-007

Anyone can selfdestruct scripts implementing delegatecall and codecall. This allows a gas griefing vector by front-running a Quark operation using such scripts and destroying the implementation, forcing users to pay more gas because the scripts bytecode should be passed as calldata and then redeployed.

Code corrected:

The script Multicall has been revised to prevent executing operations, including selfdestruct, on the context of the script itself:

```
assembly ("memory-safe") {
    thisAddress := sload(slot)
}

if (address(this) == thisAddress) {
    revert InvalidCallContext();
}
```

The variable thisAddress stores the address of the script in storage slot CONTRACT_ADDRESS_SLOT which should be set after deployment by calling function initialize().

6.8 Limited Events Emitted by Contracts

Design | Low | Version 1 | Code Corrected

CS-QUARK-008

The codebase emits only the event WalletDeploy when a new wallet is created. All other state updates do not emit any event. For instance, no event is emitted by QuarkWallet when a quark operation is executed.

It is recommended to emit events for important state updates and index the relevant parameters to allow integrators and dApps to quickly search for these and simplify UIs.

Code corrected:

The QuarkWallet contract now emits the events `ExecuteQuarkScript` when a quark operation executes successfully, and `ClearNonce` to indicate when a nonce has been cleared. We assume Compound has carefully assessed the codebase for missing events and then has decided to add only those two events.

6.9 Superfluous Allowance Given in supplyFrom

Design | Low | Version 1 | Code Corrected

CS-QUARK-009

The function `supplyFrom` in contract `CometSupplyActions` provides an allowance to `comet` from a wallet:

```
function supplyFrom(address comet, address from, address to, address asset,
    uint256 amount) external {
    IERC20(asset).forceApprove(comet, amount);
    IComet(comet).supplyFrom(from, to, asset, amount);
}
```

The allowance provided by `forceApprove()` is superfluous and not needed for the functionality.

Code corrected:

The unnecessary approval has been removed in Version 3.

6.10 Unchecked Loop Increment

Informational | Version 2 | Code Corrected

CS-QUARK-019

Since version `0.8.22`, Solidity implements the unchecked loop increment (see *Solidity changelog <https://github.com/ethereum/solidity/blob/develop/Changelog.md#0822-2023-10-25>_*) for the `for` loops of general form:

```
for (uint i = X; i < Y; ++i) {
    // variable i is not modified in the loop body
}
```

Therefore, some of the index increments that were done in an `unchecked` block can be brought back in the `for(;;)` construct to be optimized by Solidity.

Code corrected:

The index increment has been brought back in the `for(;;)` construct for loops of the general form.

6.11 Gas Griefing Through `BatchExecutor` Failure

Informational | Version 1 | Code Corrected

CS-QUARK-010

If the `BatchExecutor` is used as a centralized service to execute operations on arbitrary wallets the caller do not control, the transaction can be front-run to make the batch fail by making one of the operations fail, forcing the caller to pay the gas for a failed transaction.

Code corrected:

The contract `BatchExecutor` has been refactored and now ignores the success status of the calls from the batch, mitigating this issue.

6.12 Possible Allowance Leftover in UniswapSwapActions

Informational | Version 1 | Code Corrected

CS-QUARK-012

The function `swapAssetExactOut` gives an allowance of `amountInMaximum` to `uniswapRouter` from a wallet, however as `amountInMaximum` might be larger than the actual spent amount, there could be remaining approvals that are not fully spent.

Code corrected:

The function `swapAssetExactOut` has been updated to revoke the leftover allowed amount, if any.

6.13 Unused Code

Informational | Version 1 | Code Corrected

CS-QUARK-015

1. The imported file `QuarkWallet.sol` in `Ethcall.sol` is not used.
2. The error `CodeJar.CodeInvalid` is never used.

Code corrected:

1. The unused import has been removed.
2. The unused error has been removed.

6.14 `QuarkScript` Can Be Marked as Abstract

Informational | Version 1 | Code Corrected

CS-QUARK-016

The contract `QuarkScript` is not supposed to be deployed on its own, but rather inherited by other contracts. Therefore, this contract can be marked as `abstract` to avoid any accidental deployment.

Code corrected:

The contract `QuarkScript` has been made abstract.

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 Gas Optimizations

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

CS-QUARK-011

The codebase could be more efficient in terms of gas usage. Reducing the gas costs may improve user experience. Below is an incomplete list of potential gas inefficiencies:

1. Contract `QuarkScript` implements the function `allowCallback()` which writes the script address in storage, however no functionality to clear the storage is implemented.
2. The function `getActiveScript` performs redundant `SLOAD` operations when reading `scriptAddress`.
3. The function `setActiveNonceAndCallback` performs redundant `SLOAD` operations when reading the `scriptAddress` from mapping `nonceScriptAddress`.
4. Functions `executeQuarkOperation()` and `DOMAIN_SEPARATOR()` could be more gas efficient by caching `domainSeparator` instead of computing during each call. In that case, `domainSeparator` needs to be recomputed if `chainid` changes (e.g., due to a future fork).
5. Function `UniswapFlashLoan.run` computes redundantly the pool key via `PoolAddress.getPoolKey()`.
6. The function `setActiveNonceAndCallback` receives `ETH` and sends it back from/to the calling wallet, this transfer could be avoided by using an additional parameter in the function `setActiveNonceAndCallback` holding the `msg.value`.
7. In `QuarkWalletFactory.create()`, the assignment `executor = address(0)` is unnecessary as `executor` is zero-initialized.
8. In order to revert early, the functions `CometSupplyActions.supplyMultipleAssets`, `CometWithdrawActions.withdrawMultipleAssets`, `CometSupplyMultipleAssetsAndBorrow.run`, and `CometRepayAndWithdrawMultipleAssets.run` could check that the length of the arrays received as parameters match.
9. The wallets could be deployed behind a minimal proxy to reduce the cost of deployment.
10. During the Uniswap V3 swap callback `uniswapV3SwapCallback`, `amount0Delta` and `amount0Delta` cannot be positive at the same time.
11. The condition in `QuarkWallet.executeQuarkOperation` accepts inputs where the target address is zero and the code is empty, this will lead to the unnecessary deployment of an empty script and a no-op during the execution of the script.

Code partially corrected:

The optimizations listed above, except 1 and 4, have been implemented in the updated codebase.

1. The function `clearCallback` has been implemented but remains unused in `UniswapFlashLoan.run`.
2. The `scriptAddress` is now cached.

3. The `scriptAddress` is now cached before the callback to the wallet.
4. The optimization has not been implemented.
5. The pool key is now computed once and then reused.
6. The `ETH` transfers between the `QuarkWallets` and the `QuarkStateManager` have been removed.
7. The executor is now passed as function argument.
8. The code has been updated to check the length of the input arrays in the mentioned functions.
9. The minimal proxy pattern has been implemented.
10. The conditional branching has been updated from `if - if` to `if - else if` in order to consider only one of the amounts.
11. The function `QuarkWallet.executeQuarkOperation` has been updated to revert if the target address is 0 and the source is empty.

7.2 Minimal Proxy Cannot Be EIP-1967 Compatible

Informational **Version 2** **Acknowledged**

CS-QUARK-018

By design, the `QuarkMinimalProxy` cannot adhere to the EIP-1967 standard, that requires the `implementation` address to be stored in a precise storage slot. This is due to the fact that it must be stateless to avoid any change of implementation address by the scripts.

Note that block explorer relying on EIP-1967 will not be able to find the implementation address.

Acknowledged:

Compound responded:

```
The implementation address is set as an immutable because it 1) is not intended to
ever change and 2) lowers the gas overhead of calls that go through the proxy.
```

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 Dangerous Combinations of Scripts

Note Version 1

While taken separately, scripts can pose no security threat, but it may be the case that certain combinations of scripts, e.g. use of `multicall`, can open attack vectors. Users must be careful when using multiple scripts during one operation not to expose their wallets to new attack vectors. Developers and users should be very careful when dealing with scripts which transfer execution to untrusted addresses, e.g., native token transfer or ERC20 tokens that implement transfer hooks, validating EIP-1271 signatures, etc. Such scripts pose significant security threats and should be carefully assessed before use.

The security risks are present whenever a script passes execution to untrusted addresses. An attacker can potentially reenter the wallet and pass arbitrary parameters to the current active script that could exploit the wallet. Reentrancy guards do not always protect against such attacks.

Users should also check carefully `scriptCalldata` that are passed into a script even if the latter is known to be non-malicious. For example, passing wrong `scriptCalldata` to the script `Multicall` can have severe consequences given that it performs `delegatecalls` into arbitrary addresses.

8.2 Deprecated Opcode Callcode

Note Version 1

The design of `QuarkWallet` relies on the opcode `callcode` which has been deprecated since Solidity v0.5.0 in favour of `delegatecall`. See EIP-2488 and EIP-7. Hence, `callcode` opcode is available only through assembly code.

8.3 Executors Should Not Call Other Wallets Directly

Note Version 1

Default wallets (created with salt `0x0`) have the executor role for other wallets created for the same `signer`, hence default wallets can call `executeScript()` in other wallets. We would like to note that such calls should be done through a script like `Ethcall` which implement a normal `call`. Calling `executeScript()` directly from the default wallet (via `callcode`) or a script like Multicall (via `delegatecall`) is wrong as it would execute in the context of the default wallet.

8.4 Gas Griefing by Script `selfdestruct`

Note Version 1

Anyone can `selfdestruct` scripts implementing `delegatecall` and `codecall`. This allows a gas griefing vector by front-running a `Quark` operation using such scripts and destroying the implementation,

forcing users to pay more gas because the scripts bytecode should be passed as `calldata` and then redeployed.

The issue has been mitigated for the `Multicall` script that was in scope, but developers must still be aware of this pitfall when writing new scripts for Quark.

8.5 Payable Contracts Gas Overhead

Note **Version 5**

The two payable contracts `Paycall` and `Quotecall` define a `GAS_OVERHEAD` value that will be charged for every call to the contracts. It is supposed to cover for the transaction cost outside of the `initialGas - gasleft()` scope. The breakdown for both contracts is as follows:

- `Paycall`: `135_000 gas` total
    - `21_000 gas` for the default transaction cost
    - `9_000 gas` for the `calldata` cost. This covers between `9_000 / 16 = 562bytes` to `9_000 / 4 = 2250bytes` long calldata.
    - `70_000 gas` for the execution cost outside of the `run()` function
    - `35_000 gas` to cover for the transfer of the `payToken`
- `Quotecall`: `100_000 gas` total
    - `21_000 gas` for the default transaction cost
    - `9_000 gas` for the `calldata` cost. This covers between `9_000 / 16 = 562bytes` to `9_000 / 4 = 2250bytes` long calldata.
    - `70_000 gas` for the execution cost outside of the `run()` function

Users must be aware of the following:

1. The Quark Operations can be batched to decrease the default cost per transaction (`21_000 / N`) and increase the margin of the submitter.
2. The `70_000 gas` should slightly overpay (`5_000-8_000 gas` from Compound simulations) for a direct call to a `Wallet` compiled with `--via-ir`, without `scriptSources` (where no script is deployed).
3. In `Paycall`, the `35_000 gas` overpays (`~15_000`) in the most likely scenario of an ERC20 transfer where the sender's and the recipient's balances are non-zero initially, and slightly underpays otherwise.

8.6 Restricted Use of `CometXActions`

Note **Version 1**

The contracts `CometSupplyActions` and `CometWithdrawActions` of `TerminalScript.sol` must be used only as the targets of a `codecall()` or `delegatecall()`, and never as standalone contracts. Especially the functions `CometSupplyActions.supplyFrom`, `CometWithdrawActions.withdrawFrom`, since it would imply setting the contract as an `operator` for `from`. This would allow anyone to steal the funds of `from` addresses having the contract as `operator`.

8.7 Script Containing Callbacks Cannot Be Nested

Note Version 1

Scripts that rely on callbacks to the `QuarkWallet` do not work properly if not called directly from the `QuarkWallet`, but from another script.

This happens for the scripts `UniswapFlashLoan` and `UniswapFlashSwapExactOut`. In particular, Uniswap will callback into the `QuarkWallet` and trigger its fallback, which will then delegate call into the currently active script. In cases where the Uniswap script was called from another script (e.g. `Multicall`), the active script will not be the Uniswap script, but the caller script. Therefore, the `QuarkWallet` fallback will attempt to call the Uniswap callback inside of a script that does not posses such function, causing the entire operation to revert.

Example: `Submitter -> Wallet -> Multicall script -> UniSwapFlashLoan script`. Uniswap will callback into `Wallet`, which will delegatecall into `Multicall script`.

It is responsibility of the users to craft operations in such a way that the script callbacks are executed correctly.

8.8 Users Must Be Careful When Signing `MultiQuarkOperation`

Note Version 4

Even with EIP-712, when signing a `MultiQuarkOperation`, users do not know exactly what they are signing as they have to sign an array of hashes. But they would need to either compute the operations' hashes themselves or use some special plugin (does not exist yet) on their wallet to be able to reconstruct the hash from the operations and check that they are legit.

Signing over a malicious hash could be fatal as the operation could drain the wallet. Users must always know what they sign.

8.9 Wallets Can Call Arbitrary Scripts

Note Version 1

The contract `CodeJar` manages the deployment of new scripts via `create2`. However, the system does not restrict users from interacting with arbitrary contracts (scripts) that are deployed outside of `CodeJar`.

8.10 Wallets Should Not Execute Initialize Function in Scripts

Note Version 2

Scripts implementing a `delegate` to arbitrary addresses implement a mechanism to mitigate griefing attacks that destruct scripts, see Gas griefing through script implementation destruction. For instance, the defending mechanism in the contract `Multicall` relies on a function `initialize` which is assumed to be executed in the context of `Multicall`, i.e., `initialize()` should not be triggered via `delegatecall` or `callcode`. Therefore, users should be careful to not call

`Multicall.initialize()` in the context of wallets, otherwise the script becomes unusable by the wallet until the storage slot is cleared by another dedicated script.

8.11 Weak Reentrancy Lock

Note **Version 1**

Developers and users of quark scripts should carefully evaluate all calls that transfer execution to 3rd party accounts that are untrusted. The reentrancy guard in quark scripts has the following limitations:

1. The modifier `nonReentrant` only protects the functions protected by the `nonReentrant` modifier during the execution context of the associated nonce. For instance, in the context of nested quark operations, if only the first operation with nonce X sets the lock, only the scripts with active nonce X can be protected against reentrancy.
2. The storage location of the reentrancy lock can be overwritten by any script, bypassing the protection. E.g., in the context of a multicall, if the first script sets the lock, another script can remove the lock, allowing reentrancy in the first script.

8.12 `CodeJar` Deployment Limitations

Note **Version 1**

Users should be aware that the initialization code added to the code in `CodeJar` does not support scripts with constructor bytecode, i.e., a non-empty constructor, immutable variables, or other storage variables set during deployment. If a script happens to have constructor bytecode, this bytecode would be executed on a call instead of the intended runtime bytecode, which would break the functionality of the script. We highlight that `CodeJar` does not enforce any restriction on the bytecode that gets deployed, hence attackers can deploy through `CodeJar` scripts that behave maliciously. Users should always carefully verify the scripts they interact with and the parameters in `scriptCalldata`.

In **Version 4**, the `CodeJar` takes the full deployment bytecode as parameter instead of the runtime bytecode. This allows the deployment of contracts with a non empty constructor, lifting all the previously mentioned limitations.