PUBLIC

Code Assessment

of the Legend Scripts Smart Contracts

July 14, 2025

Produced for

by

Contents

1 Executive Summary

Dear all,

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 Legend Scripts according to Scope to support you in forming an opinion on their security risks.

Legend Labs implements Legend Scripts, a suite of scripts that enable the Quark Wallet to interact with external contracts. In particular, Legend Labs implements bridging scripts, looping scripts, wrapping scripts, and swapping scripts.

The most critical subjects covered in our audit are reentrancies, MEV, and dangling approvals. Security regarding all aforementioned subjects is high.

The general subjects covered are events, interaction with native tokens, and interoperability with common Metatransaction standards. Security regarding aforementioned subjects is high, however we have detailed the limitations of interoperability with batched operations in a note: TStoracle in bundled operations.

In summary, we find that the 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	0
Medium-Severity Findings	3
• Code Corrected	2
• Specification Changed	1
Low-Severity Findings	5
• Code Corrected	4
• Risk Accepted	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 Legend Scripts 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	9 April 2025	c9269a00f717031d6faae1f63c00401774ea8944	Initial Version
2	5 June 2025	b676704caf0ea162b32d30e929481f0503f6be1a	Updated Version
3	27 June 2025	167a1dfbd0090087c8859021b5af45f67a8b0f34	Updated Version 2
4	08 July 2025	bd0565e277f619d661ca5e8321442848902afba7	Final Version

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

The following files were in scope:

```
src/
    AaveScripts.sol
    AcrossScripts.sol
    BridgeScripts.sol
    DeFiScripts.sol
    Filler.sol
    LoopLong.sol
    LoopShort.sol
    MerklScripts.sol
    MorphoScripts.sol
    Multicall.sol
    OracleExecutor.sol
    QuotePay.sol
    SwapScripts.sol
    TStoracle.sol
    UnloopLong.sol
    UnloopShort.sol
    WrapperScripts.sol
```

The following files were already audited as part of a previous audit by ChainSecurity, and the differences with respect to commit `24617d65b06937a25ff78b77b555dff9da41e649` have been reviewed:

```
src/
    BridgeScripts.sol
    DeFiScripts.sol
    MorphoScripts.sol
    WrapperScripts.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.

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 offers scripts for the Quark Wallet, that allow performing a suite of actions on DeFi systems. For a system overview of the Quark Wallet itself, please refer to our previous reports of Quark v1 and Quark v2 .

2.2.1 Aave Script

AaveActions allows supplying liquidity to Aave to earn interest. The liquidity is not supposed to be used as collateral, therefore the reserve that is supplied is disabled as collateral (Aave automatically enables reserves as collateral when supplying). The supplying script should not be used by accounts that are currently borrowing (or intend to borrow) on Aave, since disabling the reserve as collateral can decrease its health.

Another function is provided to withdraw liquidity.

2.2.2 Across Script

The *AcrossActions* scripts allows bridging funds to other chains through the Across v3 protocol. The `depositV3()` function of the *SpokePool* of the Across protocol is called with the arguments supplied by the user, and the bridged token is pulled from the Wallet. The arguments supplied by the user are passed to the SpokePool without changes. The special flag `useNativeToken` can be used when bridging native tokens (without an ERC20 wrapper). In that case the `inputToken` needs to be specified as the wrapper contract of the native token (WETH on Ethereum Mainnet).

2.2.3 Multicall

The *Multicall* script is a script that calls other scripts. The user supplies a list of script addresses and payloads for each of the scripts (via the `run()` function). Each of the user supplied scripts is *delegatecalled*. In case of a revert, the Multicall script reverts.

The *Multicall* script also handles possible callbacks to the executing scripts: When a callback is enabled in the Quark Wallet, the currently executing script is saved in the `CALLBACK_SLOT` transient storage slot of the wallet. When the wallet is called, its `fallback()` function will forward calls to the contract in the `CALLBACK_SLOT`. In case the *Multicall* script is currently executing, the intended behavior from users is that the script that is currently called by the *Multicall* itself is the target of the callback, for this reason *Multicall* implements callback forwarding to the currently executing script, by implementing a `fallback()` function on its own.

2.2.4 QuotePay script

Signed *QuarkOperations* are expected to be relayed on-chain by Legend Labs controlled relayers such that users can interact with multiple chains without the need for native tokens on each one. The *QuotePay* script allows users to perform an arbitrary token payment to the relayer to cover the gas fees of relaying signatures.

2.2.5 Looping

Legend Scripts offers a suite of scripts to perform looping operations. Looping can be *long* or *short*.

The idea of *long* looping is to buy an *exposure* asset that has greater value than the initial *backing* asset amount in the user wallet. This is achieved by borrowing part of the backing token used to buy the exposure token and using the exposure token acquired as collateral for the borrow. After the looping, the user has a debt equal to the price paid for the exposure token acquired minus the initial backing token of the user, and has a collateral equal to the total value of the exposure token acquired.

The ratio between the collateral value and the backing token provided is the *looping factor*. If the exposure token appreciates 1%, and the looping factor is 5, the value of the user's position, which is equal to `collateral - debt`, increases by 5%.

Conversely, *short* looping consists in borrowing a greater amount of exposure token than the value of the user's initial backing token and immediately exchanging it for backing token to be used as collateral. The user therefore has a debt denominated in exposure tokens, and a collateral denominated in backing tokens. The looping factor is given by the ratio between the value of the debt and the value of the provided amount of backing token. In the *short* case, if the looping factor is 5 and the exposure token depreciates 1%, the value of the user's position appreciates by 5%.

The looping scripts achieve looping by borrowing on Morpho Blue and swapping on Uniswap V3. Uniswap V3 flash-swaps are used to buy a greater value than what is initially in the user wallet: Uniswap V3 sends the token amount (exposure token in case of long, or backing token in case of short) to the wallet, before receiving the payment for it, and calls back to the wallet in order to receive the payment. In the callback, the wallet can use the received amount as collateral for a Morpho loan that is used to pay back the swap.

Four scripts are provided:

- `LoopLong`: Opens a long looping position.
- `UnloopLong`: Closes a long looping position.
- `LoopShort`: Opens a short looping position.
- `UnloopShort`: Closes a short looping position.

To open a long looping position, `LoopLong.loop()` takes as main arguments the `exposureAmount` of `exposureToken` that the wallet will acquire and the `maxSwapBackingAmount` which is the maximum amount of `backingToken` that will be paid the users themselves in order to get `exposureAmount`. The `backingToken` provided in the swap is composed partly by funds initially in the user's wallet, up to parameter `maxProvidedBackingAmount`, and partly by funds borrowed from Morpho using the exposure token as collateral. Other parameters are the Morpho market used to borrow and the Uniswap V3 pool used to swap.

To close a long looping position, `UnloopLong.unloop()` is used by specifying the `exposureAmount` by which the collateral is reduced and `minSwapBackingAmount`, which is the minimum amount of `backingToken` received for swapping `exposureAmount`. When closing a long position, the `exposureAmount` is sold for `backingToken` and the `backingToken` is used to repay the debt.

Similarly, `LoopShort.loop()` and `UnloopShort.unloop()` open and close short positions.

The Liquidation LTV (LLTV) of Morpho markets define the maximum looping factor that can be achieved before a position becomes liquidatable. If the LLTV, for example, is 90%, the maximum looping factor is 10x (it can be computed as $\frac{1}{1-lltv}$). If the looping factor exceeds this amount, the user is liquidatable and its position can suffer a sudden drop in value when it is liquidated. The looping factor can increase because of user actions (for example calling `withdrawBackingToken()` on `LoopLong` or `LoopShort`), or because the price of the exposure token drops in case of a long / increases in case of a short. Morpho allows user to reach exactly the LLTV when opening or modifying a position, therefore users must use a margin of safety when creating or modifying their positions to stay clear of the maximum looping factor.

Users can also incur losses by slippage when swapping on Uniswap V3. The ratio between `exposureAmount` and `maxSwapBackingAmount`, or `minSwapBackingAmount` defines the limit average price at which swaps can be performed. `maxSwapBackingAmount` or `minSwapBackingAmount` should be set according to the market price of `exposureToken`.

2.2.6 TStoracle and OracleExecutor

The *TStoracle* is an on-chain key-value store where anybody can set values for a given key. The values are held in transient storage and therefore persist for the duration of one transaction. The purpose of the *TStoracle* is to pass data from the relayer of a *QuarkOperation* to the executed scripts. In general, the messages will consist of signed asset prices or swap paths to execute trades. The `put()` function of the TStoracle is unpermissioned, therefore scripts using the TStoracle should not trust the values but should implement appropriate checks to guard against malicious values.

The `OracleExecutor` contract is a convenience contract used by relayers of *QuarkOperations* to set given keys and values in the *TStoracle* before executing a *QuarkOperation* in a wallet, in the same transaction.

2.2.7 Swap Script and Filler

Contract *ApproveAndSwap* implements an intent based script to swap assets. A user signs a *QuarkOperation* to call `ApproveAndSwap.run()`, supplying as arguments `filler`, the address of the *Filler* utility contract, `sellToken` and `sellAmount`, and `buyToken` and `buyAmount`. The arguments signify an intent to buy at least `buyAmount` of `buyToken` for `sellAmount` of `sellToken`. No specific way on how to conduct the swap is specified, as it is the role of the relayer of the *QuarkOperation* to specify a swap mechanism. This is done through the Filler contract.

The *Filler* contract exposes a `swap()` function that takes as arguments `sellToken`, `buyToken`, `sellAmount` and `minBuyAmount`. It uses the TStoracle to retrieve the *swap path*, by reading the TStoracle value at the `SWAPPER_KEY` key. The swap path is encoded as a `swapContract` and a `swapData`. The Filler pulls the `sellToken` from the caller, approves the swap contract to take the `sellToken`, and calls `swapContract` with `swapData`. This is a completely untrusted external call with arbitrary data, and the filler ensures that it has received the required amount of `buyToken` after the call. Since the Filler can perform arbitrary external calls, no approvals should be given to it, except atomically just before calling the `swap` function and resetting the approval to zero afterwards.

The use of the TStoracle to obtain the swap data allows separating the concerns of the user who simply signs an intent, and of the relayer who finds a way to solve the intent and execute the *QuarkOperation*.

2.2.8 Changes in Version 3

In **Version 3** of the code, `UnloopLong` and `UnloopShort` now allow the user to fully withdraw their remaining collateral when their position has been liquidated beforehand.

2.2.9 Changes in Version 4

The `Filler` contract now allows to specify a `feeAmount` and a `feeRecipient`. The `feeAmount` can be a maximum of 1 percent of the output amount.

2.3 Trust Model

Users are responsible for signing operations that do not result in losses. Even if all scripts are initially designed to be secure, certain configurations (e.g., execution of unknown scripts or contracts that are not intended as scripts like *Filler*, uncapped max trades, insufficient slippage protection or careless bridging of tokens) can result in problematic behavior.

The `filler` address used in `ApproveAndSwap` is assumed to be a deployment of the *Filler* contract.

Relayers are partially trusted to execute operations in time (or at all).

Furthermore, all the trust assumptions in our reports of Quark v1 and Quark v2 apply here as well.

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 Open 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	1

- Enabled Callback Can Be Persisted Over Multicall Scripts Risk Accepted

5.1 Enabled Callback Can Be Persisted Over Multicall Scripts

Design Low Version 1 Risk Accepted

CS-LLS-004

Using Multicall with scripts that do not disable callbacks is unsafe, even if the scripts are safe in isolation.

Multicall keeps the status of the callback of the QuarkWallet between different sub-script executions (it does not disable the callback before executing the next script). If a script does not disable the callback, the next script could be vulnerable to unforeseen reentrancies.

It is possible that callbacks are not disabled because:

- the callback target might be optional (its execution is not guaranteed and so is `disallowCallback()`)
- the callback target might be a view function

Risk accepted:

Client states:

> We acknowledge the risk that is introduced when *disallowCallback()* is not called. In our implementation, any script that allows callbacks will also explicitly disable them within the callback.

6 Resolved Findings

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

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	3

- Missing Amount Check in Callback Code Corrected
- Multicall Reentrancy Code Corrected
- Slippage Check Race Condition Specification Changed

Low-Severity Findings	4

- Failing Native Token Transfers to Contracts Code Corrected
- Incorrect Handling of Native Tokens Code Corrected
- Loop Reentrancies Code Corrected
- Unexpected Approval Code Corrected

Informational Findings	2

- Inconsistent Events Code Corrected
- Payable Functions Not Reachable Code Corrected

6.1 Missing Amount Check in Callback

Correctness Medium Version 1 Code Corrected

CS-LLS-001

The `LoopLong` and `UnloopShort` contract's `uniswapV3SwapCallback` assume that when swapping a backing token for an exact output of exposure token, the pool will always deliver the full requested amount. It never verifies that the actual amount received equals the requested `amountOut`. In case of low-liquidity, it is possible that Uniswap V3 pools will consume all liquidity before being able to fill the full requested amount. In that case, the callback is performed with a smaller amount than requested (compare with Uniswap V3 router implementation of exactOut swaps). Therefore, it is possible that the callback receives fewer exposure token than requested, at a cost less than `maxSwapBackingAmount`. Potentially, the amount received is only a very small fraction of the requested amount, but at the cost of the full amount.

The consequence is that the slippage protection imposed by `maxSwapBackingAmount` is ineffective in defining a limit price. An attacker can exploit this to sandwich the swap.

In the same block or transaction:

1. The attacker consumes all liquidity of the pool by trading to an extreme price
2. The attacker supplies a very low amount of exposure token (`attackAmount`) at a very high price to the pool, such that purchasing `attackAmount` will cost `maxSwapBackingAmount`

3. The user performs a swap, receives `attackAmount < amountOut` and pays `maxSwapBackingAmount`
4. The attacker swaps back to initial price, recovering most of the initial capital.

Calls to `UnloopShort` are easier to exploit than calls to `LoopLong`:

`LoopLong` supplies the whole expected `exposureAmount` to the Morpho market in the callback. If the actual exposure amount received is less than `exposureAmount`, this call will fail unless the Wallet has some pre-existing `exposureToken` balance.

`UnloopShort` uses the actual received amount of `exposureToken` to repay the debt and then withdraws collateral for the whole `backingTokensOwed` amount. The repay can be of smaller magnitude than expected, and as long as the position stays above Liquidation LTV after the collateral withdrawal, the function does not revert. The attacker can tune the exploit to bring the position exactly to the Liquidation LTV, making it liquidatable in the next block as soon as interest accrues.

The cost to the attacker is only the Uniswap V3 fee required to consume all the liquidity in the pool and then trade back to normal price. This is a fixed cost, but the profit is proportional to the `exposureAmount` specified by the user.

The same issue can also arise in `LoopShort` and `UnloopLong` but it cannot be profitable for any attacker. Signing such an action for a Uniswap pool with low liquidity can, nevertheless, result in unexpected behavior.

Code corrected:

In Version 2, the `LoopLong` and `UnloopShort` scripts validate the amount received, and raise the `SwapUnderfilled` error if the amount received is lower than expected. `LoopShort` and `UnloopLong` do not validate that the full input amount has been spent in the swap, however discrepancies there can only benefit the wallet.

6.2 Multicall Reentrancy

Security | Medium | Version 1 | Code Corrected

CS-LLS-002

If the Multicall script executes a script that allows callbacks and passes execution flow to untrusted code (for example `LoopLong` with a malicious token), an attacker can execute arbitrary code in the context of the wallet. Consider the following scenario:

1. The wallet executes the multicall script, setting the active script to the multicall address.
2. The multicall script calls the `LoopLong` script.
3. `LoopLong` calls `allowCallback()`, then swaps on Uniswap using a malicious token.
4. The malicious token executes an attacker's contract.
5. The attacker contract calls back into the wallet executing the `run()` function of the multicall script
6. The wallet now delegatecalls to an attacker controlled address.

Executing malicious code through delegatecalls amounts to the loss of all funds.

Code corrected:

In Version 2, reentering the Multicall script with a call to `run()` forwards the call to the script currently executed by the Multicall script. This prevents arbitrary execution through the Multicall script.

6.3 Slippage Check Race Condition

Correctness | Medium | Version 1 | Specification Changed

CS-LLS-003

Several scripts are allowing the user to either set a `cappedMax` flag or set amount values to `type(uint256).max` in order to allow the scripts to use their current balance of a given token.

In some cases, however, this can lead to a race condition with unforeseen consequences if these values are used in conjunction with a (pre-defined) slippage check as the slippage value remains constant while the amount value is dynamic. Consider the following example:

1. A user holds 1000 USDC.
2. They sign a *QuarkOperation* which calls `AcrossScripts.depositV3()` with `cappedMax = true`, `inputAmount == uint256.max`, and an `outputAmount` of 1000 USDT on another chain.
3. After creating the message, the user creates another message that transfers 1000 USDC to their wallet.
4. The second message is executed first. The user's wallet now holds 2000 USDC.
5. The first message is executed, allowing a relayer to obtain 2000 USDC.
6. The relayer, however, is responsible for paying out only 1000 USDT on the other chain.

The following scripts are affected by this issue:

1. `AcrossScripts.depositV3()`
2. `UniswapSwapActions.swapAssetExactIn()`
3. `ApproveAndSwap.run()` (the threat is documented here though)
4. `UnloopLong.unloop()`
5. `UnloopShort.unloop()`

Specificartion changed:

`UniswapSwapActions` has been removed completely. The issue is now documented for the other functions, presenting a warning.

6.4 Failing Native Token Transfers to Contracts

Design | Low | Version 1 | Code Corrected

CS-LLS-005

`TransferActions.transferNativeToken()` uses `send()` instead of a low-level call to transfer native tokens. Transfers to contracts with fallback functions that consume more than 2300 gas are therefore not possible. Amongst others, this applies to `GnosisSafe` contracts, and even `QuarkWallet` if deployed behind a proxy.

Code corrected:

Instead of `send()`, a low level `CALL` is used with a gas limit of 10000, to be compatible with smart wallet recipients.

6.5 Incorrect Handling of Native Tokens

Correctness Low Version 1 Code Corrected

CS-LLS-006

`AcrossActions.depositV3()` allows the user to specify that their full balance of a given token should be bridged. Furthermore, it allows the user to specify whether they want to bridge an ERC-20 token or the current chain's native token. In either case, the full balance is calculated using the `balanceOf()` function of the given token and an approval is set.

The Across bridge requires users to set the `inputToken` to the respective wrapped native token contract of the current chain. If native tokens are used, the function therefore always sets an approval on the wrapped native token contract that will not be used by the bridge.

If the full balance should be used, the function uses the balance of the wrapped native token instead of the native token, which can either result in reverts or unexpected behavior.

Code corrected:

In Version 2, the amount is correctly computed for both ERC20 and native scripts, and the approval is not set in case of native token.

6.6 Loop Reentrancies

Security Low Version 1 Code Corrected

CS-LLS-007

Loop scripts call `allowCallback()` and then proceed to swap on Uniswap. In case the traded token is malicious, execution flow can be redirected to an attacker who is then able to call the script's functions that don't contain `disallowCallback()` calls.

For example, the attacker could call back to `withdrawBackingToken()` in `LoopLong` and bring the user's borrow position into a vulnerable state.

Code corrected:

Entrypoints of `LoopLong`, `LoopShort`, `UnloopLong`, `UnloopShort` have been marked as `nonReentrant` in Version 2.

6.7 Unexpected Approval

Design Low Version 1 Code Corrected

CS-LLS-008

`CometRepayAndWithdrawMultipleAssets.run()` gives an approval to a Compound contract and then calls the `supply()` function of the same contract using the same amount.

If the user passes `type(uint256).max` as the amount, the approval is set to `type(uint256).max` while the `supply()` call uses only the user's balance. This leads to an unexpected approval that is not reset back to 0 later.

Code corrected:

In Version 2, the approval is set to zero after supplying to Compound if the `repay` amount is `type(uint256).max`.

6.8 Inconsistent Events

Informational | Version 1 | Code Corrected

CS-LLS-009

The following inconsistencies of emitted events can be observed:

1. The event `TransferExecuted`, `ApproveAndSwapExecuted`,``LoopLongExecuted``, `LoopShortExecuted`, `UnloopLongExecuted`, and `UnloopShortExecuted`, `PayQuote` contain `address(this)` as a parameter. This address is always associated with the emitted event by default.
2. The event `ApproveAndSwapExecuted` is emitted with the minimum buy amount as `buyAmount`. The effective buy amount could be higher.

Legend Labs chooses to maintain the `address(this)` event parameters for explicitness. The `ApproveAndSwapExecuted` event has been removed.

6.9 Payable Functions Not Reachable

Informational | Version 1 | Code Corrected

CS-LLS-012

Some scripts define `payable` functions. However, the `QuarkWallet` has no payable entrypoints (beside the fallback function), so it is unclear what the purpose of the `payable` modifier is. The following functions are payable:

1. `WrapperActions.wrapETH`
2. `WrapperActions.wrapAllETH`
3. `WrapperActions.unwrapAllWETH`
4. `WrapperActions.wrapAllLidoStETH`
5. `AcrossScripts.depositV3`

Code corrected:

The functions are no longer `payable`.

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 Missing Refund ACKNOWELDGED

Informational **Version 1**

CS-LLS-011

`Filler.swap()` takes a `sellAmount` as input and tries to swap it for a given `minBuyAmount`. This is done using an arbitrary call. Depending on the calldata and the implementation of the target contract, it is possible that not all of the `sellAmount` is spent. The remaining amount, however, is not refunded to the user and remains on the contract where it can be transferred out by anyone (using a regular `transfer()` call and `minBuyAmount = 0`).

Client states:

> We acknowledge that not all of the sell token may be sold to fulfill the swap intent and that the unused amount will initially sit in the *Filler* contract.

7.2 Inconsistent Interface

Informational **Version 1** **Acknowledged**

CS-LLS-010

In most functions that allow this functionality, users can specify to use their full balance by setting the respective amount to `type(uint256).max` (e.g., `CometSupplyActions.supply()`).

`WrapperActions`, however, has special functions for that use case (e.g., `wrapAllETH()`).

Furthermore, `WrapperActions` contains functions for topping up the WETH balance (e.g., `wrapETHUpTo()`) but not for topping up the wstETH balance.

Client states:

> We acknowledge the inconsistent interface, but do not think any updates are warranted.

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 Compound III Default Behavior Changed

Note Version 1

In certain cases, Compound III (Comet) sets inputs equal to the user's balance if the values are set to `type(uint256).max` (e.g., `supply()` sets the amount to the current borrow balance if the base token is used).

This behavior is overridden by the new version of some of the scripts. For example, in the old version of `CometSupplyActions.supply(uint256.max)` would use the Compound III behavior, while the new version sets the supply amount to current user's balance before calling the Compound contract.

8.2 No Debt Allowed When Using Aave Scripts

Note Version 1

Users supplying to Aave using the `AaveActions` script must be aware that the `supply()` function can decrease their health factor unintentionally when their account already holds a debt position and the asset they are supplying is used as collateral for their debt position.

8.3 TStoracle in Bundled Operations

Note Version 1

`TStoracle` is used by executors to dynamically solve variables of messages signed by users. The contract only allows to set a variable once which will then persist during the execution of the whole transaction.

Since variables (`TStoracle` values) are not modifiable after being set, execution of different operations using the same `TStoracle` keys but with different values cannot be bundled in a single transaction. This includes for example bundling the execution of several QuarkOperations in a single transaction with ERC-4337, or the use of `Multicall` to perform several `SwapScript` executions.