hashlock.

Security Audit

HyperCroc (DeFi)

July 2025

info@hashlock.com.au

Hashlock Pty Ltd

Table of Contents

Executive Summary	4
Project Context	4
Audit Scope	7
Security Rating	10
Intended Smart Contract Functions	11
Code Quality	14
Audit Resources	14
Dependencies	14
Severity Definitions	15
Status Definitions	16
Audit Findings	17
Centralisation	31
Conclusion	32
Our Methodology	33
Disclaimers	35
About Hashlock	36

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

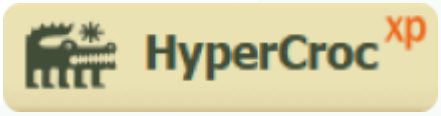
Executive Summary

The HyperCroc team partnered with Hashlock to conduct a security audit of their HyperCrocVault and Adapters smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

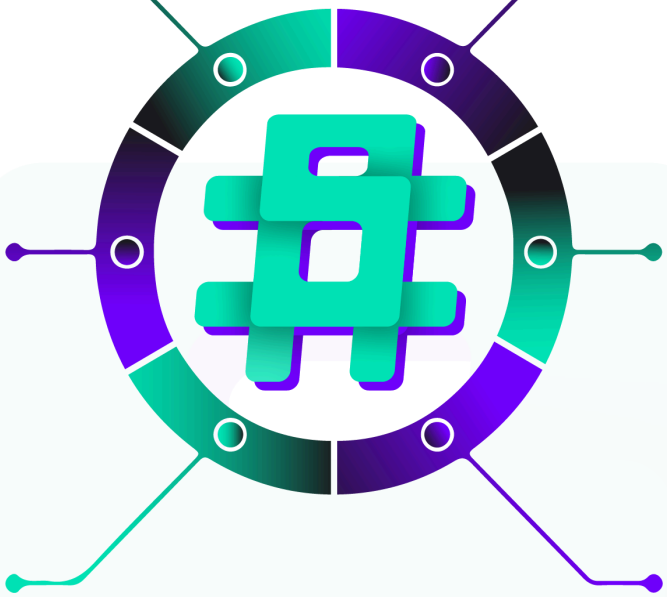
Project Context

In the swamp, the smartest Crocs hunt the highest yields. HyperCroc automates DeFi yield on HyperEVM so your capital never stops earning.

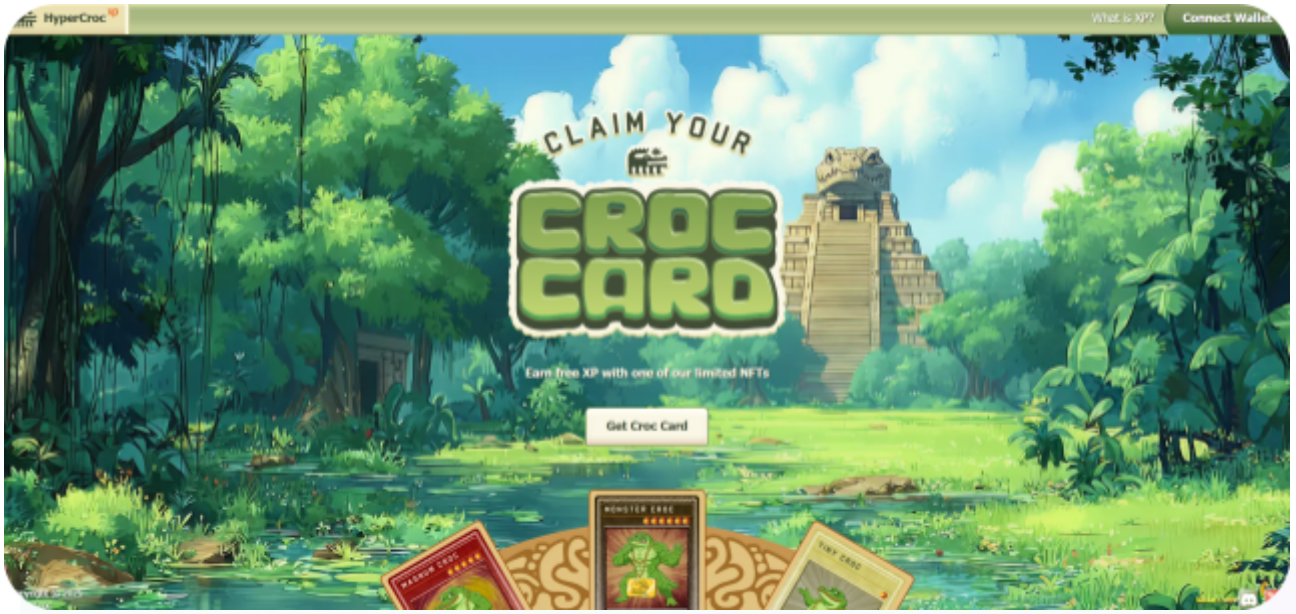
Project Name: HyperCroc
Project Type: DeFi
Compiler Version: 0.8.28
Website: https://hypercroc.xyz/
Logo:

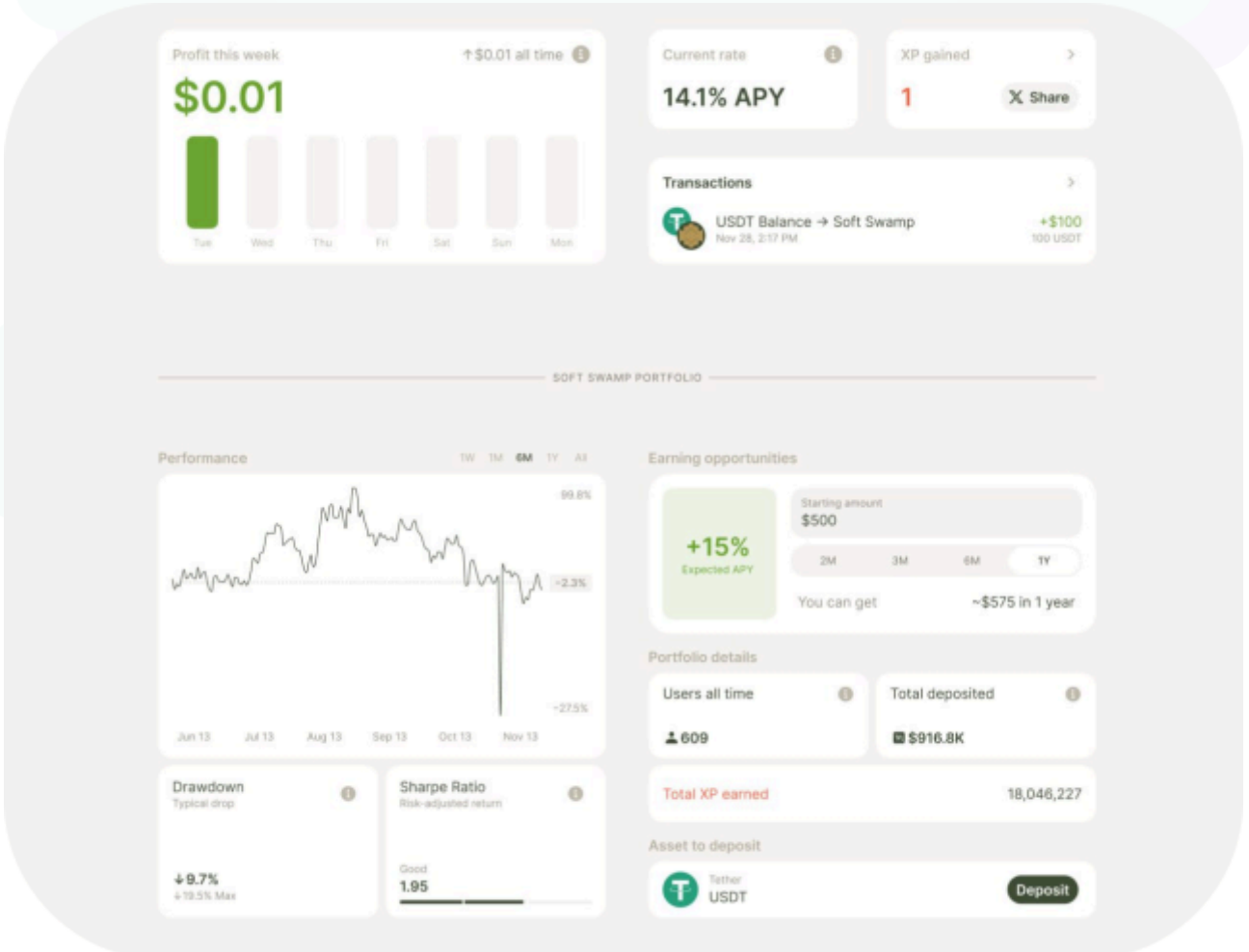


Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the solidity code within the HyperCroc project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	HyperCroc Smart Contracts
Platform	Ethereum / Solidity
Audit Date	July, 2025
Contract 1	HyperCrocVault.sol
Contract 2	HyperCrocVaultFactory.sol
Contract 3	WithdrawalQueue.sol
Contract 4	AdapterBase.sol
Contract 6	AaveAdapter.sol
Contract 7	CurveRouterAdapter.sol
Contract 8	ICurveRouterNg.sol
Contract 9	EthenaAdapter.sol
Contract 10	IStakedUSDe.sol
Contract 11	EtherfiBTCAdapter.sol
Contract 12	EtherfiETHAdapter.sol
Contract 13	IAtomicQueue.sol
Contract 14	ILayerZeroTellerWithRateLimiting.sol
Contract 15	ILiquidityPool.sol
Contract 16	IWithdrawRequestNFT.sol

Contract 17	IeETH.sol
Contract 18	IweETH.sol
Contract 19	FP96.sol
Contract 20	HyperCrocPoolAdapter.sol
Contract 21	IHyperCrocPool.sol
Contract 22	HyperCrocVaultAdapter.sol
Contract 23	LidoAdapter.sol
Contract 24	ILidoWithdrawalQueue.sol
Contract 25	IStETH.sol
Contract 26	IWstETH.sol
Contract 27	MakerDaoDaiAdapter.sol
Contract 28	MakerDaoUsdsAdapter.sol
Contract 29	MorphoAdapter.sol
Contract 30	MorphoAdapterBase.sol
Contract 31	MorphoAdapterV1_1.sol
Contract 32	IMetaMorphoFactory.sol
Contract 33	IUniversalRewardsDistributorBase.sol
Contract 34	PendleAdapter.sol
Contract 35	ResolvAdapter.sol
Contract 36	AbstractUniswapV3Adapter.sol
Contract 37	AbstractUniswapV4Adapter.sol
Contract 38	UniswapAdapter.sol
Contract 39	AdapterActionExecutor.sol
Contract 40	FactoryBase.sol
Contract 41	FeeCollector.sol
Contract 42	MultiAssetVaultBase.sol

Contract 43	OraclePriceProvider.sol
Contract 44	VaultAccessControl.sol
Contract 45	WithdrawalQueueBase.sol
Contract 46	IAdapter.sol
Contract 47	IAdapterCallback.sol
Contract 48	IEulerPriceOracle.sol
Contract 49	IExternalPositionAdapter.sol
Contract 50	IHyperCrocVault.sol
Contract 51	IHyperCrocVaultFactory.sol
Contract 52	IWETH9.sol
Contract 53	IWithdrawalQueue.sol
Contract 54	Asserts.sol
Audited GitHub Commit Hash	2ea3341c0a460f954c34132cad672e6397a5d973
Fix Review GitHub Commit Hash	aa4e161cb799077c4b4eae27ef44f481a54798a2

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section, and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

1 High severity vulnerability

4 Medium severity vulnerabilities

4 Low severity vulnerabilities

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
HyperCrocVault.sol Allows users to: - Deposit tokens into the vault - Withdraw tokens from the vault - The queue withdraws when the vault does not have sufficient tokens Allows protocol admins to: - Finalize withdrawal requests - Manage config and adapters - Add a vault manager	**Contract achieves this functionality.**
HyperCrocVaultFactory.sol Allows protocol admins to: - Deploy new HyperCroc Vaults and their associated Withdrawal Queues. - Manage the implementation contracts for the Vault and Withdrawal Queue via upgradeable beacons. - Check if a given address is an official HyperCroc Vault deployed by the factory.	**Contract achieves this functionality.**
WithdrawalQueue.sol Allows users to: - Request withdrawals of shares from the associated HyperCroc Vault, receiving an ERC721 NFT representing their request. - Claim their withdrawn assets once their withdrawal request has been finalized. Allows protocol admins to: - Finalize withdrawal requests, marking them as ready for claiming by users.	**Contract achieves this functionality.**

- Add or remove addresses authorized to finalize withdrawal requests (finalizers). - Manage the associated HyperCroc Vault address.	
libraries/Asserts.sol Provides utility functions (library) for: - Asserting that a `uint256` input is not zero, reverting with `ZeroAmount()` if it is. - Asserting that an `address` input is not the zero address, reverting with `ZeroAddress()` if it is. - Asserting that two `uint256` values are not the same, reverting with `SameValue()` if they are. - Asserting that two `address` values are not the same, reverting with `SameValue()` if they are.	**Contract achieves this functionality.**
adapters/*.sol - The adapter contracts are designed to allow the main HyperCroc Vault to interact with various external DeFi protocols and liquidity sources. They serve as intermediaries, translating the vault's generalized actions into specific calls required by each integrated protocol.	**Contract achieves this functionality.**
base/*.sol - The contracts within the `base/` directory serve as foundational building blocks for the HyperCroc Vault ecosystem, providing common functionalities and access control. They are designed to be inherited by other contracts, promoting code reusability and consistency.	**Contract achieves this functionality.**

interfaces/*.sol - The contracts within the `interface/` directory are Solidity interfaces. They define the functions and events that other contracts in the HyperCroc Vault ecosystem are expected to implement or interact with, without providing any implementation details. This ensures interoperability and defines clear communication channels between different components of the system.	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the HyperCroc project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation; however, some refactoring was recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the HyperCroc project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

High

[H-01] AbstractUniswapV3Adapter.sol - Incorrect Refund Logic in swapExactOutputV3

Description

The `swapExactOutputV3` function contains flawed logic for refunding unused tokens after a swap. The condition `if (amountIn < params.amountInMaximum - amountIn)` incorrectly restricts refunds to cases where the actual input used (`amountIn`) is less than half of the maximum allowed (`amountInMaximum`). This leaves unused tokens trapped in the contract in some scenarios and fails to reset token approvals.

Vulnerability Details

Flawed Condition:

```
if (amountIn < params.amountInMaximum - amountIn) {
    // Refund logic
}
```

Mathematically equivalent to:

`if (2 * amountIn < params.amountInMaximum)`

When `amountIn` is ≥50% of `amountInMaximum` (e.g., `amountIn = 60`, `amountInMaximum = 100`), 40 tokens remain stuck in the contract.

Proof of Concept

Scenario:

User calls `swapExactOutputV3` with:

`amountInMaximum = 100 USDC`

Swap consumes `amountIn = 60 USDC`

When the execution flow reaches the condition check:

`60 < (100 - 60) → 60 < 40 → false`

As a result, 40 USDC is not refunded to the vault, and the Uniswap router still has 40 USDC allowance.

Impact

Fund loss for users when swaps use `>50% of amountInMaximum`.

Recommendation

Make the following changes in the code:

```
function swapExactOutputV3(ISwapRouter.ExactOutputParams calldata params) external returns (uint256 amountIn) {
                if (params.recipient != msg.sender) revert WrongRecipient(msg.sender, params.recipient);

        (, address inputToken) = decodeTokens(params.path);

                IAdapterCallback(msg.sender).adapterCallback(address(this), inputToken, params.amountInMaximum);


        ISwapRouter _uniswapV3Router = uniswapV3Router;
                            IERC20(inputToken).forceApprove(address(_uniswapV3Router), params.amountInMaximum);
        amountIn = _uniswapV3Router.exactOutput(params);
++      uint256 unused = params.amountInMaximum - amountIn;
--      if (amountIn < params.amountInMaximum - amountIn) {
++      if (unused > 0) {
            IERC20(inputToken).forceApprove(address(_uniswapV3Router), 0);
--                  IERC20(inputToken).safeTransfer(msg.sender, params.amountInMaximum - amountIn);
++          IERC20(inputToken).safeTransfer(msg.sender, unused);
        }
    }
```

Status

Resolved

Medium

[M-01] WithdrawalQueue.sol - Missing ERC721Upgradeable Initialization

Description

The `WithdrawalQueue.sol` contract inherits from `ERC721Upgradeable` to manage withdrawal requests as non-fungible tokens (NFTs). However, the contract's initialize function fails to call the corresponding initializer function for this parent contract, specifically `__ERC721_init`. In the context of upgradeable smart contracts, which use a proxy pattern, the traditional constructor is replaced by an initializer function that must be explicitly called to set up the contract's initial state. The absence of this call leaves the ERC721 component of the contract, specifically `name` and `symbol` variables, uninitialized.

Vulnerability Details

The `initialize` function in `WithdrawalQueue.sol` is missing `__ERC721_init` call:

```
function initialize(address owner, address HyperCrocVault) external initializer {

   __WithdrawalQueueBase_init(owner, HyperCrocVault);

   //@eric-issue: Missing ERC721Upgradeable __ERC721_init call.

    }
```

Impact

An uninitialized `ERC721Upgradeable` contract means that critical state variables, such as the `name` and `symbol` of the NFT, are not set.

Recommendation

The `initialize` function must be updated to include a call to the `__ERC721_init` function. This will ensure that the `ERC721Upgradeable` contract is properly set up with a `name` and `symbol`.

Status

Resolved

[M-02] MultiAssetVaultBase - Unbounded Loops Can Lead to Denial of Service

Description

The `totalAssets()` function is a core component of the ERC4626 standard, designed to return the total value of all assets managed by the vault. In this implementation, the function calculates this value by iterating through an array of all `trackedAssets`, fetching their balances, querying an oracle for their prices, and summing up their value. While this logic is functionally correct, the `trackedAssets` array can grow dynamically, creating an unbounded loop.

Vulnerability Details

The vulnerability exists in the for loop within the `totalAssets` function:

```
// File: MultiAssetVaultBase.sol

function totalAssets() public view override(...) returns (uint256) {
    unchecked {
        // ... (initial balance setup)

        IERC20[] storage trackedAssets = _getMultiAssetVaultBaseStorage().trackedAssets;
        uint256 length = trackedAssets.length;

        // VULNERABLE LOOP 1
        for (uint256 i; i < length; ++i) {
            IERC20 trackedAsset = trackedAssets[i];
            uint256 trackedAssetBalance = _tokenBalance(trackedAsset);
            if (trackedAssetBalance != 0) {
                                balance  +=  _callOracle(eulerOracle,  trackedAssetBalance,
address(trackedAsset), asset);
            }
        }
        // VULNERABLE LOOP 2
        balance += _getExternalPositionAdaptersTotalAssets(eulerOracle, asset);
```

```
        return balance;
    }
}
```

The vault's owner can add new assets to the `trackedAssets` array at any time by calling the `addTrackedAsset` function. There is no limit to the number of assets that can be added.

Impact

Every transaction on the Ethereum blockchain has a gas limit per block. A transaction that consumes more gas than this limit will fail. As the owner adds more assets to the `trackedAssets` array, the total gas required to complete the loop will eventually exceed the block's gas limit. At this point, any transaction that calls the `totalAssets` function will consume all its provided gas and then revert.

Recommendation

The solution is to prevent the loop from ever becoming "unbounded." This is achieved by enforcing a sensible, finite limit on the number of assets that can be tracked.

Status

Resolved

[M-03] `AdapterActionExecutor` - Unbounded Loops Can Lead to Denial of Service

Description

`The addAdapter` function allows the contract owner to add new external position adapters. When a new adapter that implements the `IExternalPositionAdapter` interface is added, it is appended to the `externalPositionAdapters` dynamic array. There is no mechanism in place to limit the number of adapters that can be added to this array, allowing it to grow indefinitely.

Vulnerability Details

Unbounded growth of the `externalPositionAdapters` array creates a Denial of Service vector for any function that must iterate through it.

In `AdapterActionExecutor.sol`, the `_getExternalPositionAdaptersTotalAssets` function iterates through every single adapter in the `externalPositionAdapters` array to calculate the total value of assets. The gas cost of this function is directly proportional to the number of adapters in the array.

Impact

A negligent owner could repeatedly call `addAdapter` to add a very large number of adapters. As the array grows, the gas required to execute the loop inside `_getExternalPositionAdaptersTotalAssets` will also grow. Eventually, the gas cost to execute this function will exceed the block gas limit. At this point, any transaction that calls `_getExternalPositionAdaptersTotalAssets`, or any function that depends on it (like the `totalAssets` function in `MultiAssetVaultBase.sol`), will be impossible to execute successfully. They will always fail with an "out of gas" error.

Recommendation

Introduce a limit to the number of external position adapters that can be registered.

Status

Resolved

[M-04] AbstractUniswapV4Adapter.sol - Global variable block.timestamp should not be used for swap deadlines

Description

The contract uses `block.timestamp` as the `deadline` for all `_universalRouter.execute` operations without allowing users to specify their own `deadline`. This makes transactions vulnerable to being mined later than intended, potentially resulting in unfavorable swap execution due to market movements during mempool delays.

Vulnerability Details

The functions `swapExactInputV4` and `swapExactOutputV4` hardcode `block.timestamp` as the `deadline` parameter during `_universalRouter.execute` calls, universal router will execute swaps as long as `block.timestamp <= deadline`, which is always true in this implementation.

Impact

A malicious block builder can withhold swap transactions and execute them later when it's advantageous for manipulating the price or offloading tokens onto the vault at a disadvantageous price. Implementing a `deadline` parameter restricts the time frame during which an attacker can carry out such exploits.

Recommendation

Implementing a `deadline` parameter in functions `swapExactInputV4` and `swapExactOutputV4` to restrict the time frame during which an attacker can carry out such exploits.

Status

Resolved

Low

[L-01] Contracts - Unrestricted renounceOwnership() Call

Description

The contracts `FactoryBase.sol`, `FeeCollector.sol`, `VaultAccessControl.sol` and `WithdrawalQueueBase.sol` inherit from OpenZeppelin's `Ownable2StepUpgradeable` contract, which includes the `renounceOwnership()` function. This function allows the current owner to irreversibly relinquish ownership, setting the `owner` address to `address(0)`.

Impact

In the current implementation, `renounceOwnership()` is not overridden or restricted, meaning any accidental or intentional call by the owner will permanently remove owner control over the contract.

Recommendation

If ownership should never be renounced (common for fixed-governance or permanently-administered tokens), override and disable It by adding the following function:

```
function renounceOwnership() public override onlyOwner {

    revert("Renouncing ownership is disabled.");

}
```

Status

Resolved

[L-02] Contracts - Missing Address Zero Check

Description

Some functions are lacking address zero check. Specifically,

`address manager` in `VaultAccessControl.sol::addVaultManager` function.

`address finalizer` in `WithdrawalQueueBase.sol::addFinalizer` function.

Impact

It introduces ambiguity into the state of the contract, where `finalizers[address(0)]` and `_vaultManagers[address(0)]` could be true.

Recommendation

Implement address zero check in the mentioned functions.

Status

Resolved

[L-03] Contracts - Important functions lack event emissions

Description

In Solidity, events provide a critical logging mechanism that external observers (like dApps, users, or auditors) rely on to track what's happening on-chain. Important functions such as those that modify contract state, transfer funds, or change ownership should emit events to provide transparency.

The following critical functions were lacking event emissions:

```
AaveAdapter.sol::supply()

AaveAdapter.sol::_withdraw()

CurveRouterAdapter.sol::exchange()

EtherfiBTCAdapter.sol::_deposit()

EtherfiETHAdapter.sol::_deposit()

HyperCrocPoolAdapter.sol::long()

HyperCrocPoolAdapter.sol::short()

HyperCrocPoolAdapter.sol::closePosition()

HyperCrocPoolAdapter.sol::withdraw()

HyperCrocPoolAdapter.sol::emergencyWithdraw()

HyperCrocPoolAdapter.sol::_deposit()

HyperCrocVaultAdapter.sol::_deposit()

LidoAdapter.sol::_stake()

MorphoAdapterBase.sol::_deposit()

MorphoAdapterBase.sol::redeem()

MorphoAdapterBase.sol::claimRewards()

PendleAdapter.sol::swapExactTokenForPt()

PendleAdapter.sol::_swapExactPtForToken()

PendleAdapter.sol::addLiquiditySingleToken()
```

```
PendleAdapter.sol::removeLiquiditySingleToken()

PendleAdapter.sol::_redeemPt()

PendleAdapter.sol::_rollOverPt()

PendleAdapter.sol::swapTokenToToken()

PendleAdapter.sol::redeemRewards()

AbstractUniswapV3Adapter.sol::swapExactInputV3()

AbstractUniswapV3Adapter.sol::swapExactOutputV3()

AbstractUniswapV4Adapter.sol::swapExactInputV4()

AbstractUniswapV4Adapter.sol::swapExactOutputV4()

ERC4626AdapterBase.sol::_deposit()

ERC4626AdapterBase.sol::_redeem()
```

Impact

Critical actions (e.g., deposits, stakings) occur silently, making monitoring via block explorers or off-chain tools impossible.

Recommendation

Emit appropriate events in all important state-changing functions.

Status

Resolved

[L-04] MultiAssetVaultBase - Denial of Service by Blocking Tracked Asset Removal

Description

The `MultiAssetVaultBase` contract allows a privileged owner to manage a list of "tracked assets" that constitute the vault's portfolio. The `removeTrackedAsset` function is designed to allow the owner to remove an asset from this list. To ensure a clean removal, the function includes a sanity check that requires the vault's balance of the specified asset to be exactly zero. The intention is to prevent the removal of an asset that the vault still holds, which could lead to accounting errors or locked funds. However, this check inadvertently creates a denial of service vulnerability.

Vulnerability Details

The vulnerability lies in the following lines within the `removeTrackedAsset` function:

```
// File: MultiAssetVaultBase.sol

function removeTrackedAsset(address trackedAsset) external onlyOwner {
    // ... (checks for position)



    // VULNERABLE CODE BLOCK
    uint256 currentBalance = IERC20(trackedAsset).balanceOf(address(this));
    if (currentBalance != 0) revert NotZeroBalance(currentBalance);



    // ... (logic to remove asset from the array)
}
```

An attacker can exploit this logic because anyone can send ERC20 tokens to any public address, including this vault contract. The attack proceeds as follows:

1. Identify a Track Asset: An attacker identifies a `trackedAsset` in the vault that they wish to make immovable.

2. Execute the Attack: The attacker performs a standard ERC20 transfer of a minuscule, non-zero amount (e.g., 1 wei) of the tracked asset directly to the `MultiAssetVaultBase` contract's address.
3. Trigger the Vulnerability: When the legitimate owner of the vault later attempts to call `removeTrackedAsset` for the targeted asset, the balanceOf(address(this)) check will now return a non-zero value (e.g., 1). Consequently, the revert `NotZeroBalance(1)` condition will always be met.

Impact

The attack is cheap and simple to execute, making it an effective way for a malicious actor to grief the protocol and disrupt its operations without needing significant capital.

Recommendation

To maintain contract hygiene, consider adding a new privileged function that allows the owner or fee collector to "sweep" or withdraw these residual dust balances of non-tracked tokens from the contract.

Status

Acknowledged

Centralisation

The HyperCroc project values security and utility over decentralisation.

The owner executable functions within the protocol increases security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlock's analysis, the HyperCroc project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au