certora

Security Assessment
Final Report

Underscore Wallet

October 2025

Prepared for Hightop

certora

Table of content

certora

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Underscore	https://github.com/underscore-finance/underscore-protocol.git	ce62846 (original commit) 68473e8 (fix review final commit)	Vyper

Project Overview

This document describes the manual code review findings of **Underscore**. The following contract list is included in our scope:

```
contracts/core/Appraiser.vy
contracts/core/Billing.vy
contracts/core/Hatchery.vy
contracts/core/LootDistributor.vy

contracts/core/agent/AgentWrapper.vy
contracts/core/agent/SignatureHelper.vy

contracts/core/userWallet/UserWallet.vy
contracts/core/userWallet/UserWalletConfig.vy

contracts/core/walletBackpack/ChequeBook.vy
contracts/core/walletBackpack/HighCommand.vy
contracts/core/walletBackpack/Kernel.vy
contracts/core/walletBackpack/Migrator.vy
contracts/core/walletBackpack/Paymaster.vy
contracts/core/walletBackpack/Sentinel.vy
```

```
contracts/data/Ledger.vy
contracts/data/MissionControl.vy


contracts/registries/LegoBook.vy
contracts/registries/Switchboard.vy
contracts/registries/UndyHq.vy
contracts/registries/WalletBackpack.vy
```

The work was undertaken from **September 15, 2025,** to **October 13, 2025**. During this time, Certora's security researchers performed a manual audit of all the Vyper contracts and discovered several bugs in the codebase, which are summarized in the subsequent section.

Protocol Overview

The Underscore Protocol vault system wraps complex, cross-protocol yield strategies into standard ERC-4626 vaults. At the base layer are Earn Vaults., where users deposit an underlying asset such as USDC or WETH and receive shares backed by a basket of yield positions. The EarnVaultWallet module tracks balances across approved external vault tokens, computes total underlying assets and realized yield, and uses Lego adapters to deposit, withdraw, swap, and claim rewards according to rules stored in the designated vault registry. Managers or AI agents operate the vault through these adapters, but their actions are always checked against the current configuration and `MissionControl` permissions.

Leveraged Vaults sit on top of this layer and build leveraged strategies using Earn Vaults themselves as both collateral and leverage assets. A `LevgVault` accepts deposits in a base asset, then routes that capital through `LevgVaultWallet` and `LevgVaultHelper` into specific Earn Vault tokens that represent the collateral side and the borrowed-side yield positions. The contracts enforce risk controls such as a maximum debt ratio, slippage limits on GREEN and USDC flows, and structured redemption logic that can deleverage and unwind positions from Ripe and Earn Vaults before fulfilling withdrawals. As with Earn Vaults, `LevgVaults` read configuration and safety flags from VaultRegistry, rely on the same manager and agent permissioning stack, and expose a simple ERC-4626 interface to users while hiding the underlying multi-protocol mechanics.

Release Readiness - Disclaimer

Following an internal review, we believe that although the security of the protocol has been increased, the current implementation still presents a security risk.

There are three main factors contributing to this assessment:

- The volume and severity of findings to date, spread between the 2 system components - vaults and wallets, which is concerning for a release candidate.
- Ongoing development activity, including feature changes and refactoring during the audit period, outside of fixes, which has introduced new, unaudited code paths that may carry additional risk.
- Integrations with external protocols and routing to them, which cannot be fully delved into given the time-boxed review and the sheer volume of code.

Given this, with transparency in mind - we would not feel comfortable recommending that the current codebase be taken to production right after the conclusion of this audit. In our view, the protocol would still require an additional review to reach a launch-ready state, considering the amount of complex and moving parts and its volume of components and external integrations.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	6	6	6
Medium	10	10	6
Low	9	9	7
Informational	7	7	3
Total	32	32	22

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
H-01	Cheque is never deleted / deactivated after pulling	High	Fixed
H-02	There are currently no functions for letting the protocol withdraw the fees they accumulate	High	Fixed
H-03	Swaps do not check Lego permissions, allowing avoiding of fees	High	Fixed
H-04	Whitelisted payees and cheque holders could steal their respective wallet's entire funds	High	Fixed
H-05	Decimals are mishandled in multiple places in the Appraiser leading to the inflation/deflation of different values	High	Fixed
H-06	UniswapV3 Lego does not allow 1 side liquidity operations	High	Fixed
M-01	Wallet asset registration is uncapped, potentially leading to function OOGs	Medium	Fixed

M-02	Yield profits are taken based on balance differences, leading to overcharging during regular transfers between wallets	Medium	Acknowledged
M-03	Earlier participants can keep a large permanent share of future rewards	Medium	Acknowledged
M-04	When rebalancing positions to AAVE, users can skip yield profit fee payment by providing an empty `_toVaultAddr`	Medium	Fixed
M-05	Using convertToAssets to fetch share price could lead to multiple problems	Medium	Fixed
M-06	The ambassador address could be used to steal fees when user-defined	Medium	Fixed
M-07	Some actions can still be performed if the protocol is paused (user wallets)	Medium	Acknowledged
M-08	AAVE lego is missing a reward claiming mechanism	Medium	Acknowledged

M-09	Payments can be pulled while in wallet ejection mode	Medium	Fixed
M-10	UniswapV3 partial swaps could leave irrecoverable funds in the contract	Medium	Fixed
L-01	Managers can surpass the limits via transfer reentrancy	Low	Fixed
L-02	Enabled ejection mode would bypass manager value limits	Low	Fixed
L-03	Active wallet creation with active trial funds would essentially allow locking the trial funds	Low	Fixed
L-04	Migrations do not consider a wallet's cheques	Low	Fixed
L-05	Liquidity fee collection is not enabled and could lead to unclaimable fees in rare scenarios	Low	Fixed
L-06	Owner signatures should have basic sanity checks	Low	Fixed
L-07	Curve 2+ token pools disable slippage due to parameter constraints	Low	Acknowledged

L-08	failOnZeroPrice must always be set to true if there are any usdLimits being set, otherwise limits are ineffective and won't work during 0 price transactions	Low	Fixed
L-09	A manager with swap permissions can intentionally use very high slippage so that they can sandwich the transaction and steal tokens themselves	Low	Acknowledged
I-01	UniswapV2 Lego's `_getAmountIn` should check that the amountOut is not the entire reserve	Informational	Fixed
I-02	USD value is inconsistently tracked	Informational	Acknowledged
I-03	Yield withdrawing buffer for cheques is enforced inefficiently	Informational	Acknowledged
I-04	Not all Legos issue respective vault shares	Informational	Acknowledged
I-05	Unused functions	Informational	Fixed
I-06	Agent access authentication does not emit an event	Informational	Fixed

I-07	Switchboard configs are not properly cleared	Informational	Acknowledged

High Severity Issues

H-01. Cheque is never deleted / deactivated after pulling

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Billing.vy	Status: Fixed	

Description:

The Underscore wallet uses 2 payment systems: payees and cheques. Each of them are managed internally and have different configurations as they execute differently. Specifically, payees are recurring payments to specific addresses, while cheques are one-time payments to specific addresses. Each wallet sets a limit on cheques pulled per period, as payees and chequees are fully allowed to pull their own payment.
The problem here is that, the only limitations per cheque payment are the global settings for amount, but not a per-cheque validation of neither:

- Amount sent being reduced
- Any usage cheques (an IsPaid flag being switched)

This could allow cheque receivers to pull payments multiple times per allowed period, undermining other cheque recipients, since they are validated only as part of the global period and not with the intention that each cheque is a one time payment.

```
JavaScript
   @view
@external
def isValidChequeAndGetData(
    _asset: address,
    _amount: uint256,
    _txUsdValue: uint256,
    _cheque: wcs.Cheque,
    _globalConfig: wcs.ChequeSettings,
    _chequeData: wcs.ChequeData,
```

```
    _isManager: bool,
) -> (bool, wcs.ChequeData):

    # check if cheque is active
    if not _cheque.active:
        return False, empty(wcs.ChequeData)

    # check if within expiry and unlock blocks
    if block.number >= _cheque.expiryBlock or block.number < _cheque.unlockBlock:
        return False, empty(wcs.ChequeData)

    # no recipient or asset
    if empty(address) in [_cheque.recipient, _cheque.asset]:
        return False, empty(wcs.ChequeData)

    # check asset matches
    if _cheque.asset != _asset:
        return False, empty(wcs.ChequeData)

    # check amount matches cheque amount
    if _amount != _cheque.amount:
        return False, empty(wcs.ChequeData)

    # check if asset is allowed in global config
    if len(_globalConfig.allowedAssets) != 0:
        if _asset not in _globalConfig.allowedAssets:
            return False, empty(wcs.ChequeData)

    # check if USD value is zero
    if _txUsdValue == 0:
        return False, empty(wcs.ChequeData)

    # check max cheque USD value
    if _globalConfig.maxChequeUsdValue != 0:
        if _txUsdValue > _globalConfig.maxChequeUsdValue:
            return False, empty(wcs.ChequeData)

    # check if manager can pay
    if _isManager:
        if not _globalConfig.canManagerPay or not _cheque.canManagerPay:
            return False, empty(wcs.ChequeData)

    # get latest cheque data
```

```
    chequeData: wcs.ChequeData = self._getLatestChequeData(_chequeData,
_globalConfig.periodLength)

    # check pay cooldown
    if _globalConfig.payCooldownBlocks != 0:
        if block.number < chequeData.lastChequePaidBlock + _globalConfig.payCooldownBlocks:
            return False, empty(wcs.ChequeData)

    # check max num cheques paid per period
    if _globalConfig.maxNumChequesPaidPerPeriod != 0:
        if chequeData.numChequesPaidInPeriod >= _globalConfig.maxNumChequesPaidPerPeriod:
            return False, empty(wcs.ChequeData)

    # check per period paid USD cap
    if _globalConfig.perPeriodPaidUsdCap != 0:
        if chequeData.totalUsdValuePaidInPeriod + _txUsdValue >
_globalConfig.perPeriodPaidUsdCap:
            return False, empty(wcs.ChequeData)

    # update cheque data
    chequeData.numChequesPaidInPeriod += 1
    chequeData.totalUsdValuePaidInPeriod += _txUsdValue
    chequeData.totalNumChequesPaid += 1
    chequeData.totalUsdValuePaid += _txUsdValue
    chequeData.lastChequePaidBlock = block.number

    return True, chequeData
```

As it can be seen, most verification occurs globally. The only way a specific cheque is blocked is if it either expires, which does not stop repeated payments, or when they are deactivated during cancellation. Currently, the cheque payment call flow does not deactivate a used cheque.

Recommendations: Deactivate each cheque upon being pulled

Customer's response: Fixed in [f047702]

Fix Review: Fix confirmed

H-02. There are currently no functions for letting the protocol withdraw the fees they accumulate

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: LootDistributor.vy	Status: Fixed	

Description:

In the wallet system, there are currently 2 types of fees paid:

1. Transaction fees, paid during reward claiming or token swapping with the Lego integrations
2. A yield fee, paid on yield profits generated from the wallet's asset, held in DeFi protocols

Both of these paths reward 2-3 entities:

- The ambassador of the wallet, who takes a predefined cut of the whole profit, based on a percentage in a config
- *Optionally* the user, if there is a bonus asset set for them on eligible assets
- The protocol itself, which keeps the rest of the generated fees in a leftover calculation, as the rewards that are not separated for the above 2 are left for the protocol to claim.

The problem with the contract is that there is no specific functionality implemented for safely withdrawing the protocol's fees that accumulate. The only workaround method comes from the `recoverFunds` function that is inherited from the system's modules. However, this method touches all assets held in the `LootDistributor.vy`, potentially touching assets belonging to the ambassador or other user's bonuses, etc.

Recommendations: Introduce a specialized function that recovers the protocol fees, that could come with a mapping for tracking them to reduce complexity

Customer's response: Fixed in 1e896bc

Fix Review: Fix confirmed

H-03. Swaps do not check Lego permissions, allowing the avoidance of fees

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: UserWallet.vy	Status: Fixed	

Description:

Every wallet operation comes with a preset chain of validations, done through a call to `_performPreActionTasks`. The call to this validation comes with a `_shouldCheckAccess` parameter, meant to sanitize the provided Legos for interaction to make sure we are doing the correct operation. However, the `swapTokens` function sets this flag to false, allowing us to provide any DeFi Lego, not just DEXes:

```
JavaScript

def swapTokens(_instructions: DynArray[wi.SwapInstruction, MAX_SWAP_INSTRUCTIONS]) ->
(address, uint256, address, uint256, uint256):
    tokenIn: address = empty(address)
    tokenOut: address = empty(address)
    legoIds: DynArray[uint256, MAX_LEGOS] = []
    tokenIn, tokenOut, legoIds = self._validateAndGetSwapInfo(_instructions)

    # action data bundle
    ad: ws.ActionData = self._performPreActionTasks(msg.sender, ws.ActionType.SWAP, False,
[tokenIn, tokenOut], legoIds)
```

This creates an unfavourable scenario, because swaps in the wallet charge a fee only on the last asset that is swapped out, aka the `tokenOut` that comes. This opens up the following scenario:

1. We want to swap from A -> B, without paying a fee on B
2. We introduce asset C in the path, plugged as the tokenOut, so A -> B -> C
3. For the swap A -> B we provide Uniswap as the Lego for swapping
4. For the swap B -> C we provide Aave as the lego for swapping
5. The Aave lego does not implement swapping functionality, thus it returns default values

6. The tokenOut and the tokenOut amount swapped appears as 0, thus we do not pay a fee on this swap, but token B is still contained in our wallet, thus we accomplished the swap
7. We successfully dodged fees, compromising the protocol revenue

Recommendations: Turn on the lego access flag during initial validation and make sure the provided Lego is able to execute swaps. For extra safety, you could make sure the `tokenOut` from the last swap matches the initial tokenOut or that it is non-zero or both

Customer's response: Fixed in 6dbcbf1

Fix Review: Fix confirmed

H-04. Whitelisted payees and cheque holders could steal their respective wallet's entire funds

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Kernel.vy UserWalletConfig.vy	Status: Fixed	

Description:

The wallet system, apart from the payee and cheque recipient special addresses, also contains a whitelist of addresses meant to bypass any validation and instantly receive funds in case the wallet does a transfer to them. This comes with its own huge risk, managed by the manager permissions set by the wallet owner and with the mechanism where whitelisted addresses cannot be added as payees nor be given cheques, since then they will be able to pull payments and drain the wallet.

This mechanism, however, is not implemented to protect against the opposite scenario in which an address that is already a payee or already has a cheque given to them COULD be added to the list of whitelisted addresses, giving them the full capability of pulling payments without any restriction, meaning that they can essentially drain the owner's wallet.

In the most happy path, this occurrence is not so common and is a matter of configuration, however there are many entities that could act maliciously in order to achieve such an attack and the incentive depends entirely on how large the wallet is.

The following validation from the `Sentinel` on transfers shows that whitelisted addresses skip the entire process and are granted permissions to receive any amounts of funds at any interval of time:

```
JavaScript
  def _isValidPayeeAndGetData(
    _isWhitelisted: bool,
    _isOwner: bool,
```

```
    _isPayee: bool,
    _asset: address,
    _amount: uint256,
    _txUsdValue: uint256,
    _payeeConfig: wcs.PayeeSettings,
    _globalConfig: wcs.GlobalPayeeSettings,
    _payeeData: wcs.PayeeData,
) -> (bool, wcs.PayeeData):

    # whitelisted
    if _isWhitelisted:
        return True, empty(wcs.PayeeData)
```

Recommendations: The same way that whitelisted addresses cannot receive cheques and payee status, disallow payees and cheque recipients to be whitelisted to avoid total wallet drainage

Customer's response: Fixed in ce5f045

Fix Review: Fix confirmed

H-05. Decimals are mishandled in multiple places in the Appraiser leading to the inflation/deflation of different values

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Appraiser.vy	Status: Fixed	

Description:

Currently the Appraiser mishandles decimals and different type of prices in multiple places which can lead to the inflation/deflation of values:

- For 6-decimal tokens, utilizing Ripe as a price source which will return the price in 18-decimal format while share price is usually in a 6-decimal format (when convertToAssets is utilized);
    - This could lead to a great difference between the last price and the current price which would subsequently lead to inflation/deflation of wherever share price is used as calculations (yield profits calculations, for e.g.)
- If ripe price is utilized as an alternative when the share price isn't available (i.e. returns 0), it will return the actual price of the vault token, rather than the share price. For example the share price for aWETH will return the current price of WETH followed by 18 zeros.
- There are vault tokens which have 18 decimals, while the underlying token has 6 decimals which isn't taken into consideration in the Appraiser when it comes to share price calculations (e.g. uUSDC vs. USDC).
- This also goes for when calculating the share price multiplied by the underlying price, the share price can also be returned as the underlying price as well, i.e. utilizing Ripe instead of the Lego, when it returns 0.

Python

```
if config.underlyingAsset != empty(address):
            underlyingConfig: AssetUsdValueConfig =
self._getAssetUsdValueConfig(config.underlyingAsset, missionControl, legoBook, ledger)
            underlyingPrice: uint256 = self._getNormalAssetPrice(config.underlyingAsset,
underlyingConfig.legoAddr, underlyingConfig.staleBlocks, underlyingConfig.decimals)
```

```
price = underlyingPrice * pricePerShare // (10 ** underlyingConfig.decimals)
```

Recommendations: Handle all cases where decimals are mishandled by scaling them to 18 (if they're 6 decimal based), and don't mix share prices with underlying asset prices.

Customer's response: Fixed in 7fb8a01

Fix Review: Fix confirmed

H-06. UniswapV3 Lego does not allow 1 side liquidity operations

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: UniswapV3.vy	Status: Fixed	

Description:

The UniswapV3 lego allows for the use of conventional swaps and the specialized concentrated liquidity related operations, allowing wallets to earn fees on their held positions. The implementation correctly deals with swap paths, swap refunds, callbacks and liquidity position handling and NFT possession.

However, it undermines and disallows liquidity operations that deal with only 1 side of the token pair.

```
JavaScript
  def addLiquidityConcentrated(
    _nftTokenId: uint256,
    _pool: address,
    _tokenA: address,
    _tokenB: address,
    _tickLower: int24,
    _tickUpper: int24,
    _amountA: uint256,
    _amountB: uint256,
    _minAmountA: uint256,
    _minAmountB: uint256,
    _extraData: bytes32,
    _recipient: address,
    _miniAddys: ws.MiniAddys = empty(ws.MiniAddys),
) -> (uint256, uint256, uint256, uint256, uint256):
    assert not dld.isPaused # dev: paused
    miniAddys: ws.MiniAddys = dld._getMiniAddys(_miniAddys)

    # validate tokens
    tokens: address[2] = [staticcall UniV3Pool(_pool).token0(), staticcall
UniV3Pool(_pool).token1()]
```

```
    assert _tokenA in tokens # dev: invalid tokenA
    assert _tokenB in tokens # dev: invalid tokenB
    assert _tokenA != _tokenB # dev: invalid tokens

    # pre balances
    preLegoBalanceA: uint256 = staticcall IERC20(_tokenA).balanceOf(self)
    preLegoBalanceB: uint256 = staticcall IERC20(_tokenB).balanceOf(self)

    # token a
    liqAmountA: uint256 = min(_amountA, staticcall IERC20(_tokenA).balanceOf(msg.sender))
    assert liqAmountA != 0 # dev: nothing to transfer
    assert extcall IERC20(_tokenA).transferFrom(msg.sender, self, liqAmountA,
default_return_value=True) # dev: transfer failed

    # token b
    liqAmountB: uint256 = min(_amountB, staticcall IERC20(_tokenB).balanceOf(msg.sender))
    assert liqAmountB != 0 # dev: nothing to transfer
    assert extcall IERC20(_tokenB).transferFrom(msg.sender, self, liqAmountB,
default_return_value=True) # dev: transfer failed

......
......
......

@external
def removeLiquidityConcentrated(
    _nftTokenId: uint256,
    _pool: address,
    _tokenA: address,
    _tokenB: address,
    _liqToRemove: uint256,
    _minAmountA: uint256,
    _minAmountB: uint256,
    _extraData: bytes32,
    _recipient: address,
    _miniAddys: ws.MiniAddys = empty(ws.MiniAddys),
) -> (uint256, uint256, uint256, bool, uint256):
    assert not dld.isPaused # dev: paused
    miniAddys: ws.MiniAddys = dld._getMiniAddys(_miniAddys)

    # make sure nft is here
    nftPositionManager: address = UNIV3_NFT_MANAGER
```

```
    assert staticcall IERC721(nftPositionManager).ownerOf(_nftTokenId) == self # dev: nft not
here

    # get position data
    positionData: PositionData = staticcall
UniV3NftPositionManager(nftPositionManager).positions(_nftTokenId)
    originalLiquidity: uint128 = positionData.liquidity

    # validate tokens
    tokens: address[2] = [positionData.token0, positionData.token1]
    assert _tokenA in tokens # dev: invalid tokenA
    assert _tokenB in tokens # dev: invalid tokenB
    assert _tokenA != _tokenB # dev: invalid tokens

    # organized the index of tokens
    minAmount0: uint256 = _minAmountA
    minAmount1: uint256 = _minAmountB
    if _tokenA != tokens[0]:
        minAmount0 = _minAmountB
        minAmount1 = _minAmountA

    # decrease liquidity
    liqToRemove: uint256 = min(_liqToRemove, convert(positionData.liquidity, uint256))
    assert liqToRemove != 0 # dev: no liquidity to remove

    params: DecreaseLiquidityParams = DecreaseLiquidityParams(
        tokenId=_nftTokenId,
        liquidity=convert(liqToRemove, uint128),
        amount0Min=minAmount0,
        amount1Min=minAmount1,
        deadline=block.timestamp,
    )
    amount0: uint256 = 0
    amount1: uint256 = 0
    amount0, amount1 = extcall
UniV3NftPositionManager(nftPositionManager).decreaseLiquidity(params)
    assert amount0 != 0 and amount1 != 0 # dev: no liquidity removed
```

As it can be seen in the code for adding and removing liquidity, the positions that are currently in range of the price would be handled correctly, as we are forced to provide non-zero amounts when adding and to receive non-zero amounts when receiving.

But for positions that are outside of the range, due to price movement, we will be completely unable to add more liquidity or to even recover our funds, exactly due to the code above reverting when 1 side of the token pair is empty.

The scenarios are 2, 1 of them is a DoS to addition until the position moves into range, if it ever does again, the other is a complete lock up of user funds, either temporarily or permanently depending on market conditions:

1. Alice adds liquidity to the [1500; 2500] price range and is forced to provide both token A and token B due to the current price, respective to the price at the bounds she chose
2. The price moves outside of her range, converting all of our liquidity into only token B, as token A got sold out of that specific range
3. Now any time we wish to add liquidity, we will be forced to provide only token B to satisfy the UniswapV3 proportions. However, the Lego forces us to provide amounts for both tokens, otherwise the transaction will revert us early. We are in a deadlock
4. If we decide we want to retrieve our position's funds, we want to remove it, burn the LP tokens and get our respective share of token Bs, since the token A we provided was swapped out. However the code forces the Lego to receive some amount in both tokens. This leads to another deadlock, amplified by a loss of funds, as our funds become stuck in UniswapV3 until we reenter the price range. This is optimistic and could never occur again, making the loss permanent.

Recommendations: Do not enforce the token amounts when providing or receiving to be non-zero. Any mistakes on the proportions made by the user are their mistake and UniswapV3 will revert internally. For removal, when burning an LP position it's possible for token0 or token1 to be a 0 amount, so that check should be removed entirely as well

Customer's response: Fixed in 82cc455

Fix Review: Fix confirmed

Medium Severity Issues

M-01. Wallet asset registration is uncapped, potentially leading to function OOGs

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: UserWalletConfig.vy Billing.vy LootDistributor.vy Hatchery.vy Migrator.vy	Status: Fixed	

Description:

The easiest way to introduce new tokens to a userWallet is through swaps, or by directly sending them to the wallet. This list of tokens held by each wallet is fully iterated on several operations throughout the wallet, like: withdrawing from yield opportunities for payments, clawing back trial funds, claiming assets in the loot contract, migrating from one wallet to another, manually updating asset configurations in the `WalletConfig` contract.

These iterations have 2 potential paths for exploit and maliciously causing an OOG error, rendering those operations undoable:

1. If a malicious manager has no restrictions in terms of assets, they can swap small wei amounts of placeholder tokens as many times as they want where `_performPostActionTasks` would register each and every one of those assets through `_updateAssetData` call. This is easily done through attempting swaps on Lego's whose swap implementations return empty amounts, as those pass the transaction as successful and still registers the assets as held in the wallet.
2. A malicious wallet creator could set their ambassador as the victim they want to target for reward claiming DoS. Every swap operation and yield generation event on the malicious wallet would also pay a percentage to the ambassador, which is directly reflected in their `claimableLoot`

Both of these scenarios, while config dependent in some cases, could result in hundreds, or even thousands of registered assets.

A lower cost scenario for exploiting the swaps to bubble up asset counts for both the owner and ambassador victim is:

1. Creating a pool on one of the Lego DEXes that supports some arbitrarily created asset and load it with some liquidity themselves

2. Swapping through the wallet to generate an amount that is enough to accumulate a tiny fee and thus get added to the ambassador's asset list

3. Rinse and repeat until the ambassador's claimableAssets list is too large for them to claim due to OOG

Costs here are very rough, since we have the different chains and managers are more limited by their periods and allowed swap amounts, as well as some smaller tx costs. The ambassador however loops over all assets, accesses storage for every single asset, every loot, calls to `balanceOf`, approvals, logs, external calls to themselves for receiving the assets, external calls to Ripe, initial permission checks, with tx caps of around 30M for ETH mainnet and the currently used Base, it would require <1000 assets to cause a DoS.

Recommendations: While manager exploitations are config dependent, they could be limited via capping the amount of assets a manager can interact with during a period. For the ambassador targeted DoS, the free choice of an ambassador is strongly recommended to be removed in place for some system for determining the ambassador, if allowed

Customer's response: Fixed in 5fac65e

Fix Review: Fix confirmed

M-02. Yield profits are taken based on balance differences, leading to overcharging during regular transfers between wallets

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: UserWallet.vy Appraiser.vy	Status: Acknowledged	

Description:

The Underscore wallet charges a yield profit fee on its yield assets in either:

- The value difference that comes from a positive increase in the exchange rate
- The increase in token balance caused by a rebasing event

The latter is wrongly implemented for assets whose config marks them as rebasing, as this difference of the current vs the last saved balance is too objective. The balance increase that the system might detect as yield could be the result of a simple transfer of tokens between 2 wallets, for e.g:

1. Alice has a balance of 100_000 aUSDC
2. A rebasing event happens, increasing Alice's balance to 100_010 aUSDC
3. Bob sends her 10_000aUSDC for whatever reason
4. The system would detect the change in Alice's balance as 10_010 and charge the fee with that amount as a parameter

This is partially mitigated by the Appraiser's `_handleRebaseYieldAsset`:

```
JavaScript
    def _handleRebaseYieldAsset(
     _currentBalance: uint256,
     _lastBalance: uint256,
     _maxYieldIncrease: uint256,
     _performanceFee: uint256,
```

```
) -> (uint256, uint256, uint256):

    # no profits if balance decreased or stayed the same
    if _lastBalance == 0 or _currentBalance <= _lastBalance:
        return 0, 0, 0

    # calculate the actual profit
    uncappedProfit: uint256 = _currentBalance - _lastBalance
    actualProfit: uint256 = uncappedProfit

    # apply max yield increase cap if configured
    if _maxYieldIncrease != 0:
        maxAllowedProfit: uint256 = _lastBalance * _maxYieldIncrease // HUNDRED_PERCENT
        actualProfit = min(uncappedProfit, maxAllowedProfit)

    return 0, actualProfit, _performanceFee
```

The partial handling comes from putting a cap on the possible fee taken from the delta amount, but this only limits losses and does not eliminate them. The current implementation would lead to us almost always hitting the maximum profit cap, since there is no way to differentiate between which tokens came from the rebasing event and which came from the balance transfers.

Recommendations: Due to this scenario having a high likelihood, it would be best to just keep the cap tied, unless a more complex tracking logic is introduced

Customer's response: Acknowledged

M-03. Earlier participants can keep a large permanent share of future rewards

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: LootDistributor.vy	Status: Acknowledged	

Description:

The Underscore wallet tracks a metric in the form of USD value of each operation, using it to periodically issue rewards to users, which gather points based on exactly this USD value of their actions. The `LootDistributor` is the contract doing the updates and the `Ledger` holds the data, which is updated every block and represents a share of the current deposit reward asset:

JavaScript

```
   def _claimDepositRewards(
    _user: address,
    _ripeTeller: address,
    _ripeLockDuration: uint256,
    _ledger: address,
) -> uint256:
    self._updateDepositPoints(_user, 0, False, _ledger)

    # get user and global points
    userPoints: PointsData = empty(PointsData)
    globalPoints: PointsData = empty(PointsData)
    userPoints, globalPoints = staticcall Ledger(_ledger).getUserAndGlobalPoints(_user)
    if userPoints.depositPoints == 0 or globalPoints.depositPoints == 0:
        return 0

    # check if there is anything available for rewards
    data: DepositRewards = self.depositRewards
    if data.asset == empty(address) or data.amount == 0:
        return 0

    # calculate user's share, transfer to user
```

```
    availableRewards: uint256 = min(data.amount, staticcall
IERC20(data.asset).balanceOf(self))
    userRewards: uint256 = availableRewards * userPoints.depositPoints //
globalPoints.depositPoints
    if userRewards == 0:
        return 0
```

As it can be seen, the reward is a percentage of the total available reward asset that is periodically loaded inside the contract.

The current logic achieves correct short terms rewards, but greatly undermines later system participants, due to the linear increase in both global points and per user points and the lack of incentive for reward claiming, instead of holding.

The scenario is as following:

1. We deploy 100_000$ worth of rewards and let users accumulate points
2. The system is still young, so there are not a lot of global points and participants, thus Alice manages to accumulate points that represent 10-15-20% of the current allocation
3. Alice does not claim those rewards. Everybody else claims, but due to the proportions, Alice's allocation is still in the contract until claimed
4. Some time passes and the team deployed 1_000_000$ worth of rewards to increase participation incentives yet again. The newer users engage with the system, but their higher count causes their personal points to be much greatly lower than the global, due to the sheer higher number of participants
5. Alice, however, still has her points, which used to represent 10% of the global. With the new participants, this number could have dropped 1-2%, but this would still allow Alice to claim a large portion of the new rewards, without having to interact with the wallet at all when they got added

As it can be seen, the system is not currently fit to handle holding of points and it creates an unfair advantage for earlier participants.

Recommendations: Multiple mitigations are possible, depending on how much code is wished to be revamped: auto-claiming rewards for users, so they can restart their points; separate

reward epochs, tracked in a mapping; a Synthetix-like reward mechanism that tracks rewards per token and allows adding more rewards, without biasing on time in the system

Customer's response: Acknowledged

M-04. When rebalancing positions to AAVE, users can skip yield profit fee payment by providing an empty `_toVaultAddr`

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: UserWallet.vy	Status: Fixed	

Description:

- User calls `rebalanceYieldPosition` and provides an empty `_toVaultAddr` (This is plausible if rebalancing to AAVE) from any lego;
- In all other cases (other legos) the `_toVaultAddr` is returned as the toToken (i.e. the 4626 vault which would be the same as the to yield token)

```
JavaScript
   def rebalanceYieldPosition(
    _fromLegoId: uint256,
    _fromVaultToken: address,
    _toLegoId: uint256,
    _toVaultAddr: address = empty(address),
    _fromVaultAmount: uint256 = max_value(uint256),
    _extraData: bytes32 = empty(bytes32),
) -> (uint256, address, uint256, uint256):
    ad: ws.ActionData = self._performPreActionTasks(msg.sender, ws.ActionType.EARN_REBALANCE,
False, [_fromVaultToken, _toVaultAddr], [_fromLegoId, _toLegoId])
```

- When `_performPreActionTasks` is called the assets will contain the `_fromVaultToken` and the `_toVaultAddr`, with the second one being empty;
- After withdrawing from the source lego, the flow will continue to deposit to the "destination" lego.
- Within `_depositForYield` :

Python

```
def _depositForYield(
    _asset: address,
    _vaultAddr: address,
    _amount: uint256,
    _extraData: bytes32,
    _shouldPerformPostActionTasks: bool,
    _shouldGenerateEvent: bool,
    _ad: ws.ActionData,
) -> (uint256, address, uint256, uint256):
    amount: uint256 = self._getAmountAndApprove(_asset, _amount, _ad.legoAddr) # doing
approval here

    ## deposit for yield
    assetAmount: uint256 = 0
    vaultToken: address = empty(address)
    vaultTokenAmountReceived: uint256 = 0
    txUsdValue: uint256 = 0
    assetAmount, vaultToken, vaultTokenAmountReceived, txUsdValue = extcall
Lego(_ad.legoAddr).depositForYield(_asset, amount, _vaultAddr, _extraData, self,
self._packMiniAddys(_ad.ledger, _ad.missionControl, _ad.legoBook, _ad.appraiser))
    self._resetApproval(_asset, _ad.legoAddr)
```

A call to the `depositForYield` on the AAVE Lego will be made:

Python

```
vaultToken: address = self._getVaultTokenOnDeposit(_asset, _vaultAddr, miniAddys.ledger,
miniAddys.legoBook)

      ## Then we have the actual call which determines the vaultToken:

      def _getVaultTokenOnDeposit(_asset: address, _vaultAddr: address, _ledger: address,
_legoBook: address) -> address:
    vaultToken: address = yld.assetOpportunities[_asset][1] # aave v3 only has one
opportunity
    isRegistered: bool = True

    # not yet registered, call aave directly to get vault token
    if vaultToken == empty(address):
        vaultToken = self._getAaveVaultToken(_asset, self._getPoolDataProvider())
        isRegistered = False
```

```
assert vaultToken != empty(address) # dev: invalid vault token

# make sure input matches
if _vaultAddr != empty(address):
    assert vaultToken == _vaultAddr # dev: vault token mismatch

# register if necessary
if not isRegistered:
    self._registerAsset(_asset, vaultToken)
    self._updateLedgerVaultToken(_asset, vaultToken, _ledger, _legoBook)

return vaultToken
```

The vaultToken will be fetched from configs, and the input validation won't be performed due to having an empty _vaultAddr. Then vaultToken will be returned which would allow for further deposit.

Recommendations:

Customer's response: Fixed in 34adac4

Fix Review: Fix confirmed

M-05. Using convertToAssets to fetch share price could lead to multiple problems

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: legos/yield/.	Status: Fixed	

Description:

When the yield tax is gathered on non-rebasing assets, those that fetch the share prices, in all legos, we're using `convertToAssets` in order to get the share price, but the problem with this method is that convertToAssets doesn't get the user-specific share price, but the average one.

First it's not inclusive of fees (in some cases), and if the protocol has any fees during withdrawals (and there are protocols which have), the users will pay the yield tax with those fees included.

So users won't pay the yield tax on the actual amount withdrawn to the protocol but the share difference amount which isn't the net amount but the gross one as it might include fees.

- Moonwell's `exchangeRateStored` doesn't account for accrued interest;

Recommendations: Preview the redemption value, which ideally will be inclusive of the underlying protocol's fees

Customer's response: Fixed in 309f216

Fix Review: Fix confirmed

M-06. The ambassador address could be used to steal fees when user-defined

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: Hatchery.vy	Status: Fixed	

Description:

Each wallet has 2 creation paths, based on the system's configuration, either:

- Creation is restricted to only special addresses, which set the parameters for each new wallet
- Wallet creation is permissionless, allowing user-provided parameters

The second path would also lead to the user being able to provide any wallet address to be their `ambassador` and earn a percentage of the swap fees and yield profit fees. While meant to incentivize protocol widespread and onboarding of new users, this permissionless path would also allow users to provide their own wallets for ambassadors and instead capture this percentage of fees, essentially getting a discount on fees they pay to the system.

JavaScript

```
   def createUserWallet(
    _owner: address = msg.sender,
    _ambassador: address = empty(address),
    _shouldUseTrialFunds: bool = True,
    _groupId: uint256 = 1,
) -> address:
    assert not deptBasics.isPaused # dev: contract paused
    a: addys.Addys = addys._getAddys()

    config: UserWalletCreationConfig = staticcall
MissionControl(a.missionControl).getUserWalletCreationConfig(msg.sender)
    assert config.startingAgent != _owner # dev: starting agent cannot be the owner

    # validation
    if not addys._isSwitchboardAddr(msg.sender):
```

```
        assert config.isCreatorAllowed # dev: creator not allowed
    assert empty(address) not in [config.walletTemplate, config.configTemplate, _owner] #
dev: invalid setup
    if config.numUserWalletsAllowed != 0:
        assert staticcall Ledger(a.ledger).numUserWallets() < config.numUserWalletsAllowed #
dev: max user wallets reached

    # ambassador
    ambassador: address = empty(address)
    if _ambassador != empty(address) and staticcall
Ledger(a.ledger).isUserWallet(_ambassador):
        ambassador = _ambassador
```

Recommendations: Like in M-01, either whitelist a set of ambassadors, or make the parameter non-user defined and make the setting of an ambassador to a wallet be a permission action, executed based on off-chain conditions, like invite link generation, etc.

Customer's response: Fixed in 5fac65e

Fix Review: Fix confirmed

M-07. Some actions can still be performed if the protocol is paused

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: UserWallet.vy	Status: Acknowledged	

Description:

Contrary to the pausing mechanism's purpose, if DeptBasics is paused the Appraiser will just return 0, but that won't always stop the transaction from being executed, which won't serve the purpose of stopping transfers, deposits, swaps, etc.

In multiple Appraiser functions we see the following (or similar) code:

```
JavaScript
   # if paused, fail gracefully
    if deptBasics.isPaused:
        return 0, 0, 0
```

But returning a 0, won't always cause a failure of the user wallet transaction, for e.g. if `failOnZeroPrice` is set to true, the function execution will continue.

Besides this, other functions within Appraiser which return 0 in case of a pause (like the example given above) which is located in calculateYieldProfits, won't terminate the execution process, as:

```
Python
 feeRatio: uint256 = 0
    data.lastPricePerShare, yieldRealized, feeRatio = extcall
Appraiser(_ad.appraiser).calculateYieldProfits(_asset, currentBalance, data.assetBalance,
data.lastPricePerShare, _ad.missionControl, _ad.legoBook)
```

```
# only save if appraiser returns a price per share (non-rebasing assets)
if data.lastPricePerShare != 0:
    self.assetData[_asset] = data

# pay yield fee
self._payYieldFee(_asset, yieldRealized, feeRatio, _ad)

# mark as checked
self.checkedYield[_asset] = True
```

Recommendations: Introduce a pausing mechanism which when enabled can enact a pause on all wallets, OR freeze all wallets at once through the .isFrozen check.

Customer's response: Acknowledged

M-08. AAVE lego is missing a reward claiming mechanism

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: AaveV3.vy	Status: Acknowledged	

Description:

Depending on the underlying protocol behind it, each Lego introduces custom logic for its respective operations. In the case of yield strategy protocols, there's logic for claiming rewards, generated on some basis. These rewards are most usually tokens, that can be claimed.
AAVE is one of the protocols that includes incentive rewards for holding aTokens and a merit program that airdrops rewards to users whose actions "align with the strategic objectives of AAVE".

However, the implementation to claim any of the rewards is missing:

```
JavaScript
def claimRewards(
    _user: address,
    _rewardToken: address,
    _rewardAmount: uint256,
    _extraData: bytes32,
    _miniAddys: ws.MiniAddys = empty(ws.MiniAddys),
) -> (uint256, uint256):
    return 0, 0
```

This means that any reward generated for the wallet contract is unclaimable, leading to a loss of value for the owner.

Recommendations: Introduce respective reward claiming for AAVE rewards, based on reward distribution events

Customer's response: Acknowledged

M-09. Payments can be pulled while in wallet ejection mode

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: UserWallet.vy Billing.vy	Status: Fixed	

Description:

When eject mode is set by the switchboard (admin), it's set with the intention of being an emergency withdrawal solution, in which only transfers and eth-to-weth, as well as vice-versa conversions should be allowed.

The first problem is that payees can still pull payments themselves, and not just owner/manager being able to transfer funds, while cheques are disallowed inside `preActionTasks`, which deviates from the intention of safely moving the wallet's asset, for example during migration.

The second aspect of this problem is that the protocol will charge yield profit fees under certain conditions, while in eject mode, when this shouldn't be the case, as it is strictly avoided throughout the other parts where eject mode is verified:

During payment pulling if not enough is present within the contract, the `_withdrawFromYieldOpportunities` flow will be invoked and marked as a special transaction.

From here, we will bypass the `_performPreActionTasks` mechanism which reverts if tx type is not a transfer or weth-to-eth and vice versa mechanism, and allow the yield withdrawing call, to take any profits on share price changes or token rebalances, breaking the core intention of the mechanism:

JavaScript

```
   def _validateCanTransfer(
    _signer: address,
    _recipient: address,
    _asset: address,
    _isSpecialTx: bool,
    _isCheque: bool,
) -> (address, ws.ActionData):
    ad: ws.ActionData = empty(ws.ActionData)
    assert _recipient != empty(address) # dev: inv recipient

    # finalize asset
    asset: address = _asset
    if asset == empty(address):
        asset = ETH

    # only wallet config can do trusted txs (migration, clawback trial funds)
    if _isSpecialTx:
        walletConfig: address = self.walletConfig
        assert _signer == walletConfig # dev: perms
        ad = staticcall WalletConfig(walletConfig).getActionDataBundle(0, _signer)
        self._checkForYieldProfits(asset, ad)

    # normal transaction
    else:
        action: ws.ActionType = ws.ActionType.TRANSFER
        if _isCheque:
            action = ws.ActionType.PAY_CHEQUE
        ad = self._performPreActionTasks(_signer, action, False, [asset], [], _recipient)

    return asset, ad

....
....
....

def _performPreActionTasks(
    _signer: address,
    _action: ws.ActionType,
    _shouldCheckAccess: bool,
    _assets: DynArray[address, MAX_ASSETS],
    _legoIds: DynArray[uint256, MAX_LEGOS] = [],
    _transferRecipient: address = empty(address),
) -> ws.ActionData:
```

```
    legoId: uint256 = 0
    if len(_legoIds) != 0:
        legoId = _legoIds[0]
    ad: ws.ActionData = staticcall
WalletConfig(self.walletConfig).checkSignerPermissionsAndGetBundle(_signer, _action, _assets,
_legoIds, _transferRecipient)

    # cannot perform any actions if wallet is frozen
    assert not ad.isFrozen # dev: frozen wallet

    # eject mode can only do transfer and eth conversions
    if ad.inEjectMode:
        assert _action in (ws.ActionType.TRANSFER | ws.ActionType.ETH_TO_WETH |
ws.ActionType.WETH_TO_ETH) # dev: invalid action in eject mode
        return ad
```

Recommendations: Specifically disable payee payments if the intention is for sole migrations and disable yield redemption to avoid charging fees during an emergency

Customer's response: Fixed in 6f69c40

Fix Review: Fix confirmed

M-10. UniswapV3 partial swaps could leave irrecoverable funds in the contract

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: UniswapV3.vy	Status: Fixed	

Description:

The `swapTokens()` function of the UniswapV3 lego verifies and goes over the entire token path to swap the tokens, based on the user's instructions and does 2 verifications:

- If only a partial amount of tokenIn was used, refund the rest
- The regular slippage check for the tokenOut

These are both correct, as UniswapV3 has the possibility of executing "partial" swaps, meaning that if the liquidity of the traded pair runs out, the swap instruction will stop and return only the amount that was successfully swapped.
While this scenario occurs only in lower liquidity pools and needs to satisfy the tokenOut slippage check, it is still a plausible scenario in the case of AI agent, that might choose the lower liquidity pools as more optimal for the paths.

The problem arises due to the fact that partial swaps could occur in the middle of the path, for e.g: A -> B -> C, where some of token B gets leftover. This would have been fine if it is leftover in the Lego contract, as it can be retrieved. However the `_swapTokensInPool` function hardcodes the amount to be swapped, and uses it during the callback, instead of calculating the amount to send the pool, via the returned deltas from UniswapV3:

```
JavaScript
    def _swapTokensInPool(
     _pool: address,
     _tokenIn: address,
     _tokenOut: address,
```

```
    _amountIn: uint256,
    _recipient: address,
    _uniswapV3Factory: address,
) -> uint256:
    tokens: address[2] = [staticcall UniV3Pool(_pool).token0(), staticcall
UniV3Pool(_pool).token1()]
    assert _tokenIn in tokens # dev: invalid tokenIn
    assert _tokenOut in tokens # dev: invalid tokenOut
    assert _tokenIn != _tokenOut # dev: invalid tokens

    assert staticcall UniV3Factory(_uniswapV3Factory).getPool(_tokenIn, _tokenOut, staticcall
UniV3Pool(_pool).fee()) == _pool # dev: invalid pool

    self.poolSwapData = PoolSwapData(
        pool=_pool,
        tokenIn=_tokenIn,
        amountIn=_amountIn,
    )
#..... rest of the function

@external
def uniswapV3SwapCallback(_amount0Delta: int256, _amount1Delta: int256, _data: Bytes[256]):
    poolSwapData: PoolSwapData = self.poolSwapData
    assert msg.sender == poolSwapData.pool # dev: no perms

    # transfer tokens to pool
    assert extcall IERC20(poolSwapData.tokenIn).transfer(poolSwapData.pool,
poolSwapData.amountIn, default_return_value=True) # dev: transfer failed
    self.poolSwapData = empty(PoolSwapData)
```

This means that if only ⅔ of token B's amount was used up, we will send the entirety of the B amount to the V3 pool, leaving it unrecoverable.

Recommendations: Utilize the `_amount0Delta` and `_amount1Delta` to determine how much tokens to send the pool and recover the rest of the stuck tokens. A positive delta is the amount we owe the pool and a negative delta is the amount we automatically receive

Customer's response: Fixed in be003e1

Fix Review: Fix confirmed

Low Severity Issues

L-01. Managers can surpass the limits via transfer reentrancy		
Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: UserWallet.vy	Status: Fixed	

Description:

The current wallet implementation offers both pre and post transaction validation checks for the value of the transaction and the entity that executed it, specifically the wallet managers, which are bound by value limits, period limits, cooldowns, tx counts, etc.

The order in which these are applied are:

- The pre checks sanitize the number of transactions and the cooldown of the manager
- The post checks the USD value of the operation

This discrepancy between the timing of sanity checks opens up the window for a reentrancy bypass inside of `transferFunds:`

```
JavaScript
   def transferFunds(
    _recipient: address,
    _asset: address = empty(address),
    _amount: uint256 = max_value(uint256),
    _isCheque: bool = False,
    _isSpecialTx: bool = False,
) -> (uint256, uint256):
    asset: address = empty(address)
    ad: ws.ActionData = empty(ws.ActionData)
    asset, ad = self._validateCanTransfer(msg.sender, _recipient, _asset, _isSpecialTx,
_isCheque)

    # finalize amount
    amount: uint256 = 0
```

```
    if asset == ad.eth:
        amount = min(_amount, self.balance)
    else:
        amount = min(_amount, staticcall IERC20(asset).balanceOf(self))
    assert amount != 0 # dev: no amt

    # get usd value
    txUsdValue: uint256 = self._updatePriceAndGetUsdValue(asset, amount, ad)

    # make sure recipient can actually receive funds
    if not _isSpecialTx:
        if _isCheque:
            assert extcall WalletConfig(ad.walletConfig).validateCheque(_recipient, asset,
amount, txUsdValue, ad.signer) # dev: cheque invalid
        else:
            assert extcall
WalletConfig(ad.walletConfig).checkRecipientLimitsAndUpdateData(_recipient, txUsdValue,
asset, amount) # dev: recipient limits exceeded

    # do actual transfer
    if asset == ad.eth:
        send(_recipient, amount)
    else:
        assert extcall IERC20(asset).transfer(_recipient, amount, default_return_value =
True) # dev: xfer

    self._performPostActionTasks([asset], txUsdValue, ad, _isSpecialTx)
    log WalletAction(
        op = 1,
        asset1 = asset,
        asset2 = _recipient,
        amount1 = amount,
        amount2 = 0,
        usdValue = txUsdValue,
        legoId = 0,
        signer = ad.signer,
    )
    return amount, txUsdValue
```

As it can be seen, there are no reentrancy protections in place. In the case of tokens with a callback, a manager can reenter as many times as he wants when transferring out tokens, as long as he does not surpass his total USD limit. What he gets to bypass is:

1. The number of TXs he should be allowed to do in a period
2. The cooldown he should have, which becomes meaningless

Recommendations: Add a reentrancy guard to protect against this and further potential issues

Customer's response: Fixed in 41579ed

Fix Review: Fix confirmed

L-02. Enabled ejection mode would bypass manager value limits

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: UserWallet.vy	Status: Fixed	

Description:

The `ejectMode` is a special wallet flag that, when turned on, disables all operations except transfers and conversions between ETH and WETH. This is a feature, mostly for migrations, however it comes with the edge-case where the transaction USD value, used to limit managers, is set to 0 intentionally inside the post transaction checks. This would allow managers to freely transfer funds, without ever hitting their limit

Recommendations: Do not disable USD values for ejection transactions so managers can be validated, disable only the unnecessary actions that use the USD value

Customer's response: Fixed in 6f69c40

Fix Review: Fix confirmed

L-03. Active wallet creation with active trial funds would essentially allow locking the trial funds

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: Hatchery.vy	Status: Fixed	

Description:

The wallet creation is protected behind a whitelist config, that either allows permissionless deploying of wallets or only protocol addresses can deploy. The trial funds logic also depends on being enabled for new wallets and takes them from a pre-loaded balance. However, if both of these configs are enabled, we would essentially be able to deploy as many wallets with the purpose of emptying this trial funds reserve. While they cannot be stolen, they become essentially useless.

Recommendations: There is no algorithmic way to mitigate this targeted wallet creation, except for enforcing the trial funds to no be freely given out to new wallets

Customer's response: Fixed in 1447296

Fix Review: Fix confirmed

L-04. Migrations do not consider a wallet's cheques

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: Migrator.vy	Status: Fixed	

Description:

The wallet migration process goes through a set of validations for both the new and the old wallet, between which the transfer is happening. Both wallets have to:

- Both are not frozen
- The caller is the owner of both
- The receiving wallet has no payees and no whitelisted address, no managers
- There is no pending owner change

As it can be seen, cheques are nowhere to be found. We do not cheque their existence or lack there of on both ends, neither do we transfer the configs for cheques from the old to the new wallet, like we do with the other specific settings:

```
JavaScript
   # 2. copy all managers (except starting agent)
    managersCopied: uint256 = 0
    numManagers: uint256 = staticcall UserWalletConfig(fromConfig).numManagers()
    if numManagers > 1:
        for i: uint256 in range(1, numManagers, bound=max_value(uint256)):
            manager: address = staticcall UserWalletConfig(fromConfig).managers(i)
            if manager == empty(address):
                continue

            # skip the starting agent from source wallet
            if manager == fromStartingAgent:
                continue

            managerSettings: wcs.ManagerSettings = staticcall
UserWalletConfig(fromConfig).managerSettings(manager)
            if managerSettings.startBlock != 0:
```

```
                extcall UserWalletConfig(toConfig).addManager(manager, managerSettings)
                managersCopied += 1

    # 3. copy global payee settings
    globalPayeeSettings: wcs.GlobalPayeeSettings = staticcall
UserWalletConfig(fromConfig).globalPayeeSettings()
    extcall UserWalletConfig(toConfig).setGlobalPayeeSettings(globalPayeeSettings)

    # 4. copy all payees
    payeesCopied: uint256 = 0
    numPayees: uint256 = staticcall UserWalletConfig(fromConfig).numPayees()
    if numPayees > 1:
        for i: uint256 in range(1, numPayees, bound=max_value(uint256)):
            payee: address = staticcall UserWalletConfig(fromConfig).payees(i)
            if payee == empty(address):
                continue

            payeeSettings: wcs.PayeeSettings = staticcall
UserWalletConfig(fromConfig).payeeSettings(payee)
            if payeeSettings.startBlock != 0:
                extcall UserWalletConfig(toConfig).addPayee(payee, payeeSettings)
                payeesCopied += 1

    # 5. copy all whitelisted addresses
    whitelistCopied: uint256 = 0
    numWhitelisted: uint256 = staticcall UserWalletConfig(fromConfig).numWhitelisted()
    if numWhitelisted > 1:
        for i: uint256 in range(1, numWhitelisted, bound=max_value(uint256)):
            addr: address = staticcall UserWalletConfig(fromConfig).whitelistAddr(i)
            if addr != empty(address):
                extcall UserWalletConfig(toConfig).addWhitelistAddrViaMigrator(addr)
                whitelistCopied += 1
```

While they can be added manually, this is missing functionality and could lead to cheques on the new wallet executing during the migration process, disrupting it.

Recommendations: Add the necessary migration checks for cheques as well

Customer's response: Fixed in [fdc858d]

Fix Review: Fix confirmed

L-05. Liquidity fee collection is not enabled and could lead to unclaimable fees in rare scenarios

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: UniswapV3.vy	Status: Fixed	

Description:

The Uniswap fees that get accumulated inside our position get collected only when adding or removing liquidity from our own position. However, it is completely possible for a regularly managed position to be emptied out without collecting its fees. It is also completely plausible to assume that our wallet could receive and hold such a position in some form of event or exchange, but it will have no way to claim those sitting fees, as we are forced to increase the position to force a collection. The `_collectFees` function has only an internal implementation:

```
JavaScript
   def _collectFees(_nftPositionManager: address, _tokenId: uint256, _recipient: address,
_positionData: PositionData) -> (uint256, uint256):
    if _positionData.tokensOwed0 == 0 and _positionData.tokensOwed1 == 0:
        return 0, 0
    params: CollectParams = CollectParams(
        tokenId=_tokenId,
        recipient=_recipient,
        amount0Max=max_value(uint128),
        amount1Max=max_value(uint128),
    )
    return extcall UniV3NftPositionManager(_nftPositionManager).collect(params)
```

Recommendations: Introduce a simple fee collection function to account for the edge-case where a wallet holds an empty position with fees uncollected

Customer's response: Fixed in afb3218

Fix Review: Fix confirmed

L-06. Owner signatures lack basic sanity checks

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: AgentWrapper.vy	Status: Fixed	

Description:

Currently, an agent's owner can freely execute any action, bypassing any signature checks:

```
JavaScript
    def _authenticateAccess(_userWallet: address, _messageHash: bytes32, _sig: Signature):
     owner: address = ownership.owner
     if msg.sender != owner:
         # check expiration first to prevent DoS
         assert _sig.expiration >= block.timestamp # dev: signature expired

         # check nonce is valid
         assert _sig.nonce == self.currentNonce[_userWallet] # dev: invalid nonce

         # verify signature and check it's from owner
         signer: address = self._verify(_messageHash, _sig)
         assert signer == owner # dev: invalid signer

         # increment nonce for next use
         self.currentNonce[_userWallet] += 1
```

This is usually fine, but in the case the owner of the agent provides a valid signature, unknowingly, this signature can later be replayed, potentially maliciously, since it burns no nonce nor is verified at all.

Recommendations: Do basic sanity to make sure the owner did not accidentally provide a valid check or force the owner to omit the `_sig` parameter to avoid mistakes

Customer's response: Fixed in 441fe88

Fix Review: Fix confirmed

L-07. Curve 2+ token pools disable slippage due to parameter constraints

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: Curve.vy	Status: Acknowledged	

Description:

The Curve AMM pools have the capability of holding more than 2 tokens for exchanging. However, our abstract Lego interface allows for providing only the standard 2 tokens when providing liquidity. This is fine when we also give the specific pool we target, but it disallows us from setting the slippage we expect on the token we are meant to receive during liquidity removal:

JavaScript

```
   def _removeLiquidityTricrypto(
    _p: PoolData,
    _lpAmount: uint256,
    _minAmountA: uint256,
    _minAmountB: uint256,
    _recipient: address,
) -> (uint256, uint256):
    minAmountsOut: uint256[3] = [0, 0, 0]
    minAmountsOut[_p.indexTokenA] = _minAmountA
    minAmountsOut[_p.indexTokenB] = _minAmountB

    # NOTE: user can only specify two min amounts out, the third will be set to zero

    # remove liquidity
    amountsOut: uint256[3] = extcall TriCryptoPool(_p.pool).remove_liquidity(_lpAmount,
minAmountsOut, False, _recipient, False)
    return amountsOut[_p.indexTokenA], amountsOut[_p.indexTokenB]
```

In the above example of 3 token pools, our 3rd token's slippage is completely disabled, allowing unfavourable for the user amounts to pass through.

Recommendations: Use the `_extraData` parameter to allow the caller to encode the 3rd and 4th token's slippage parameters and decode them if they are expected to allow more flexibility and fairness to the caller.

Customer's response: Acknowledged

L-08. failOnZeroPrice must always be set to true if there are any usdLimits being set, otherwise limits are ineffective and won't work during 0 price transactions

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: Paymaster.vy	Status: Fixed	

Description:

When setting different limits and restraints in the wallet, there's no hard rule imposed that `failOnZeroPrice` must be true, if we're setting the usd limits to a certain Payee.

This means that the limits would be ineffective if the transaction value returns 0, but we have `failOnZeroPrice` set to `false` as the transaction won't revert.

The transaction value can be returned as 0, due to a number of reasons including tokens which don't have an oracle pricing (e.g. Governance tokens), current problems with the oracle source, or an oracle-reported price of 0.

Whenever we impose USD limits, the `failOnZeroPrice` has to be set to true.

Recommendations: Impose a rule in which if any USD limits are being set `failOnZeroPrice` can't be false.

Customer's response: Fixed in 82cc455

Fix Review: Fix confirmed

L-09. A manager with swap permissions can intentionally use very high slippage so that they can sandwich the transaction and steal tokens themselves

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: UserWallet.vy	Status: Acknowledged	

Description:

The wallet doesn't have any mechanism which automatically would check slippage against some hardcoded parameters or limit it in any way. Since wallet owners can allow any manager/agent to "control" wallet operations, this means that malicious managers/agents can intentionally set very high slippage to the swap, in order to "sandwich" the transaction.

Considering that the manager will "sandwich" their own transaction, there's no need to frontrun in the traditional sense, but they can decide when to execute it and in which way.

Recommendations: Calculate slippage atomically to make sure that it's within certain bounds.

Customer's response: Acknowledged

Informational Issues

I-01. UniswapV2 Lego's `_getAmountIn` should check that the amountOut is not the entire reserve

Description:

The `_getAmountIn()` function of the lego, used internally for view-only operations checks amount in for a given amount out by checking the reserves, but it does not validate that the amount out is not the entire reserve of the given pool, which would cause a division by 0 revert.

```
JavaScript
    def _getAmountIn(_pool: address, _zeroForOne: bool, _amountOut: uint256) -> uint256:
     if _amountOut == 0 or _amountOut == max_value(uint256):
         return max_value(uint256)

     reserveIn: uint256 = 0
     reserveOut: uint256 = 0
     reserveIn, reserveOut = self._getReserves(_pool, _zeroForOne)
     if reserveIn == 0 or reserveOut == 0:
         return max_value(uint256)

     if _amountOut > reserveOut:
         return max_value(uint256)

     numerator: uint256 = reserveIn * _amountOut * 1000
     denominator: uint256 = (reserveOut - _amountOut) * 997
     return (numerator // denominator) + 1
```

Recommendations: Handle the entire reserve swap case

Customer's response: Fixed in 441fe88

Fix Review: Fix confirmed

I-02. USD value is inconsistently tracked

Description:

The wallets track their token balances' total value denominated in USD, playing major roles in determining user deposit rewards and impacting some event emissions, since there is no way to track direct ERC20 token transfers to the wallet. Updates to the value happen only on wallet operations, so there is a high possibility of unrecorded balance sitting in a more dormant wallet.

Recommendations: Implement an automatic wallet update or choose a different tracking method for reward generation

Customer's response: Acknowledged

I-03. Yield withdrawing buffer for cheques is enforced inefficiently

Description:

The 1% buffer added to yield withdrawing, when cheque transfer amounts are insufficient, is implemented as a hard add-on that must be covered when yield withdrawing instead of a range to enter from 100% to 101%. This means that in the case where we have withdrawn 100.5% of the needed funds and we cover the cheque, we will continue looping before we reach the end of the assets and pay the respective amount, thus burning through gas and overlooping.

Recommendations: Ease up the buffer check in the case where a lot of assets are expected to be looped, since a couple of missing wei could cause gas burn

Customer's response: Acknowledged

I-04. Not all Legos issue respective vault shares

Description:

While the yield withdrawal for cheques involves withdrawing from the underlying legos, the Ripe lego is a special case that's not handled, since it does not issue shares to the wallet. This means that there could be a scenario where our Ripe position could cover a cheque payment, but we cannot access them during redemption, ultimately failing the payment

Recommendations: Determine if the Ripe lego should suffice to cheque payments and handle its case

Customer's response: Acknowledged

I-05. Unused functions

Description:

The `LootDistributor.vy` contract implements an unused `updateDepositPointsOnEjection` function, probably left over from tests or meant for future updates.

Recommendations: Clear out unused functions to free up contract size or mark them for future implementations

Customer's response: Fixed

Fix Review: Fix confirmed

I-06. Agent access authentication does not emit an event

Description:

While the `AgentWrapper.vy` includes an event emission in its manual `incrementNonce` function, the same emission is missing in the `_authenticateAccess` function which does all of the validations for signatures when a legit transaction comes through.

Recommendations: Emit consistent events when important state changes

Customer's response: Fixed in 441fe88

Fix Review: Fix confirmed

I-07. Switchboard configs are not properly cleared

Description:

When switchboard actions get executed/cancelled, only the `actionType[_aid]` entry gets cleared up and not the entire pending action itself. While execution itself is fine, as it is guarded by the type, leftover pending entries can be viewed off-chain and could cause confusion.

Recommendations: Clear the pending action itself too, not only the type correlating to the action id

Customer's response: Acknowledged

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.