hashlock.

Security Audit

Balloon (DeFi)

September 2025

info@hashlock.com.au

Hashlock Pty Ltd

Table of Contents

Executive Summary	4
Project Context	4
Audit Scope	7
Security Rating	8
Intended Smart Contract Functions	9
Code Quality	10
Audit Resources	10
Dependencies	10
Severity Definitions	11
Status Definitions	12
Audit Findings	13
Centralisation	33
Conclusion	34
Our Methodology	35
Disclaimers	37
About Hashlock	38

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

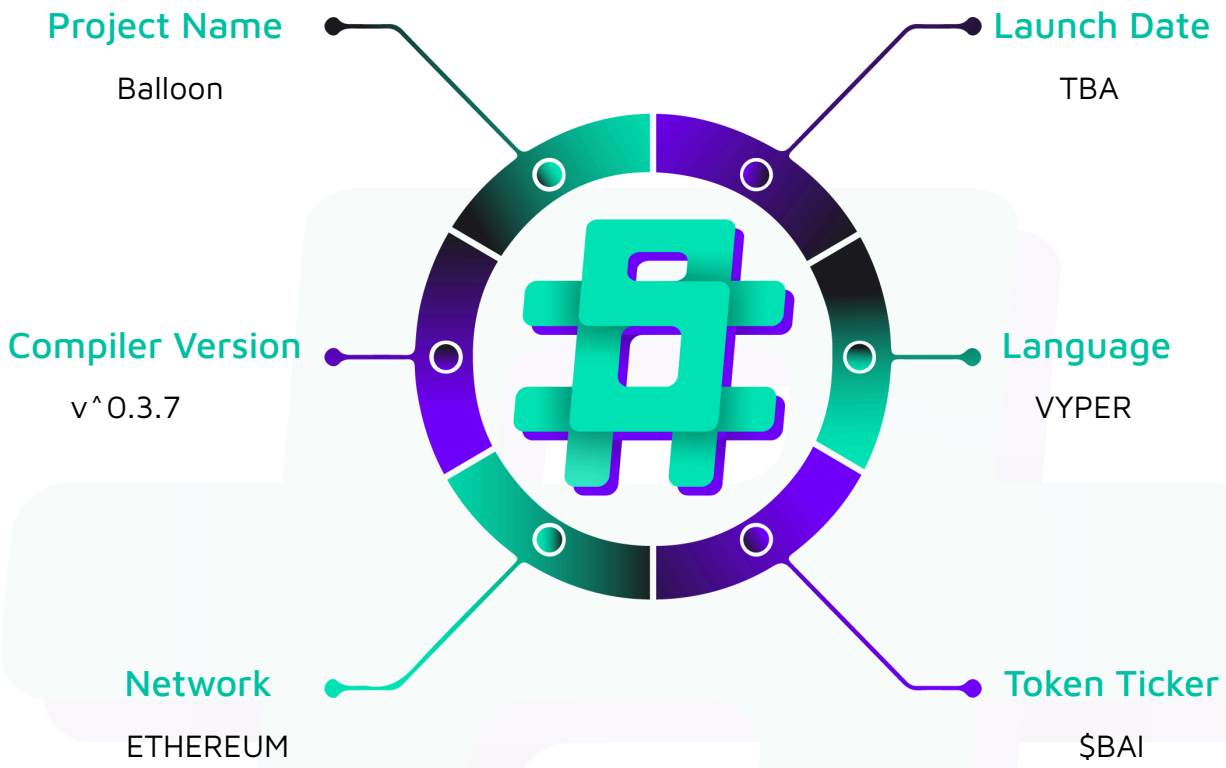
The Balloon team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

Balloon Tech is a platform that develops DeFAI Agents, autonomous artificial intelligence systems that operate nonstop across multiple blockchain networks to optimize yield farming opportunities. These agents are designed to analyze real-time data, identify the best rates, and automatically allocate assets into curated vaults such as ETH-USDC, Gold, Blackrock, and BTC. By combining AI-driven strategies with human oversight, BalloonTech aims to maximize returns while reducing inefficiencies in decentralized finance. For example, their ETH-USDC vault currently offers competitive APYs around 18.2%, showcasing how automation and intelligence can create sustainable advantages in DeFi.

Project Name: Balloon
Project Type: DeFi
Compiler Version: ^0.3.7
Website: https://www.balloontech.ai/
Logo:

Visualised Context:



Project Visuals:

Audit Scope

We at Hashlock audited the Vyper code within the Balloon 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	**Balloon Smart Contracts**
Platform	**Ethereum / Vyper**
Audit Date	**September, 2025**
Contract 1	Vault.vy
Contract 2	VaultFactory.vy
Audited GitHub Commit Hash	107dc58f85daad0215c8d3954b7e21226b351c22
Fix Review GitHub Commit Hash	89a6147583824f7fae0a4d7aa6dfbf130fd3f403

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.

Hashlock found:

4 Medium severity vulnerabilities

3 Low severity vulnerabilities

2 Gas Optimisations

1 QA

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
Vault.vy - Allows users to: - Stake ERC20 tokens for specified lock periods - Earn rewards based on lock duration and amount - Withdraw staked tokens and rewards after the lock period - Reset their stake to compound rewards - Allows admins to: - Set reward rates for different lock periods - Recover ERC20 tokens from the contract - Manage admin roles	**Contract achieves this functionality.**
VaultFactory.vy - Allows users to: - Deploy their own ERC4626-compliant vaults with custom parameters - View protocol fee configurations for vaults - Allows admins to: - Set and update protocol fee rates and recipients - Configure custom protocol fees for specific vaults - Remove custom protocol fees - Shutdown the factory to prevent new vault deployments - Manage and transfer governance roles	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Balloon 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 Balloon 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

Medium

[M-01] Vault#permit - Signature Malleability in Permit Function

Description

The `_permit` implementation in the Vault contract is vulnerable to signature malleability. Due to the lack of restrictions on the s value in the ECDSA recovery process, two distinct signatures `((v, r, s)` and `(v', r, s'))` can be generated for the same signed message. While nonce validation prevents replay attacks, this divergence from best practices introduces off-chain inconsistencies and undermines signature uniqueness guarantees expected in `EIP-2612-compliant` implementations.

Vulnerability Details

The `_permit` function uses `ecrecover` directly without enforcing the canonical form of the s value. In Ethereum's `secp256k1` curve, both `s` and `(n - s)` are valid solutions, which leads to two valid signatures for the same digest. Most standard implementations, such as OpenZeppelin's `EIP-2612`, enforce the lower-half s value to mitigate malleability.

```
assert ecrecover(

    digest, v, r, s

) == owner, "invalid signature"
```

Because no check ensures s lies in the lower half of the curve order, any valid signature can be transformed into an alternate valid signature. This does not enable replay attacks due to nonce incrementation, but it breaks signature determinism and can confuse off-chain systems that rely on unique signatures for logging, auditing, or indexing.

Proof of Concept

The following Foundry test demonstrates that both the original and a malleated signature can successfully execute a permit:

```
function testSignatureMalleabilityPermit() public {

    // --- Setup and Preconditions ---

    // Generate test addresses and private key for 'bob' (owner) and 'alice' (spender)

    (address bob, uint256 bobPrivateKey) = makeAddrAndKey("bob");

    address alice = makeAddr("alice");

    // Construct the EIP-2612 permit digest for the signature

    bytes32 digest = keccak256(

        abi.encodePacked(

            "\x19\x01",

            vault.DOMAIN_SEPARATOR(),

            keccak256(

                abi.encode(

                    keccak256("Permit(address owner,address spender,uint256 value,uint256
nonce,uint256 deadline)"),

                    bob,

                    alice,

                    type(uint256).max,

                    0,

                    type(uint256).max

                )

            )

        )

    );
```

```
    // Sign the digest with bob's private key to get (v, r, s)
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(bobPrivateKey, digest);

    // Compute the malleated signature (v', s') such that both are valid
                                                uint256            secp256k1n            =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141;
    uint256 s_int = uint256(s);
    uint256 s_int_prime = secp256k1n - s_int;
    bytes32 s_prime = bytes32(s_int_prime);
    uint8 v_prime;
    if (v == 27) {
        v_prime = 28;
    } else if (v == 28) {
        v_prime = 27;
    } else {
        v_prime = (v == 0) ? 1 : 0;
    }
    // --- Exploit Execution ---
    // Use the original signature to call permit
    uint256 snapshot = vm.snapshotState();
    vault.permit(bob, alice, type(uint256).max, type(uint256).max, v, r, s);
     assertEq(vault.allowance(bob, alice), type(uint256).max, "Both signatures should set
max allowance");
    // Revert state to before the first permit, then use the malleated signature
    vm.revertToState(snapshot);
    vault.permit(bob, alice, type(uint256).max, type(uint256).max, v_prime, r, s_prime);
```

```
    // --- Post-condition Check ---

    // Both signatures should have successfully approved the allowance

    assertEq(vault.allowance(bob, alice), type(uint256).max, "Both signatures should set max allowance"); // PoC succeeds if both signatures set allowance

}
```

Impact

Although replay is prevented by nonce handling, the existence of multiple valid signatures for the same message can disrupt off-chain monitoring, signature indexing, or compliance tools expecting determinism. This divergence from standard `EIP-2612` implementations may also reduce protocol composability with third-party tooling that expects strict canonical signatures.

Recommendation

Introduce a validation check to ensure the s value lies in the lower half of the `secp256k1` curve order, consistent with `EIP-2` requirements and OpenZeppelin's implementation. This enforces signature uniqueness and eliminates malleability:

```
require(uint256(s) <= 0x7fffffffffffffffffffffffffffffff5d576e7357a4501ddfe92f46681b20a0, "invalid s value");

require(v == 27 || v == 28, "invalid v value");
```

This ensures only canonical signatures are accepted.

Status

Resolved

[M-02] Vault - Missing Reward Claim Functionality

Description

The Vault contract allows deposits into external `ERC4626` strategy contracts, which may provide additional incentive rewards (e.g., Morpho's incentive pool). However, the current implementation does not include any mechanism to call reward-claiming functions (such as `claimRewards`). As a result, accrued rewards remain locked in the strategy and are never retrieved. This omission leads to a permanent loss of incentive yield for users.

Vulnerability Details

The root cause is that the Vault contract only implements deposit and withdrawal functions for strategies but lacks support for invoking reward-related functions. Many yield strategies require explicit reward collection via function signatures like `claimRewards` or similar. Without the ability to trigger these calls, the Vault cannot realize the full return of underlying strategies. Thus, reward tokens generated by strategies accumulate but remain inaccessible. Since function signatures differ across protocols, a rigid implementation is insufficient. A flexible call mechanism is required to safely allow claiming.

Impact

Users of the Vault lose all potential incentive rewards provided by underlying strategies. Over time, this can result in a significant reduction in effective APY compared to direct deposits into the strategy. The Vault therefore, provides worse returns than expected, undermining trust and usability of the system.

Recommendation

Introduce a mechanism to claim rewards from strategies. Two approaches are recommended:

1. Implement a generic `executeCall` function restricted to the `Vault owner`, allowing controlled invocation of arbitrary external functions.
2. Alternatively, maintain a whitelist of reward function selectors at deployment, and allow the Vault to call these functions safely.

Status

Resolved

[M-03] Vault#_redeem - Loss Skew from Strategy Selection in Withdrawals

Description

The Vault contract allows users to specify a custom withdrawal queue of strategies when redeeming or withdrawing assets. This creates a scenario where users can selectively avoid strategies with heavy unrealized losses, leaving those losses to be realized by later withdrawers. Additionally, even when losses are unavoidable, users who withdraw earlier and carefully select their strategy order will suffer less loss compared to those who withdraw later. As a result, unrealized losses are not distributed fairly, and opportunistic users can game the system to minimize their share of losses.

Vulnerability Details

The vulnerability originates in the `_redeem` function, which supports passing a strategies array. If provided, and `use_default_queue` is false, the vault withdraws from the user-selected strategies rather than the full default queue:

```
if len(strategies) != 0 and not self.use_default_queue:

    _strategies = strategies
```

Since the calculation of unrealized losses relies on `_assess_share_of_unrealised_losses`, only the selected strategies are checked. This means:

1. A user can completely bypass lossy strategies by excluding them.
2. A user who withdraws earlier in a lossy strategy queue will experience a smaller proportional loss than those withdrawing later.

This leads to unfair distribution of losses across participants and creates an incentive for users to rush withdrawals when losses are suspected.

Impact

Unrealized losses are not shared proportionally among all participants. Instead, early and selective withdrawers minimize or avoid losses, while late withdrawers disproportionately absorb them. This breaks fairness, encourages racing behavior, and can lead to significant wealth transfer from passive users to fast-acting opportunistic ones. The systemic effect is erosion of trust and potential value drain in stressed conditions.

Recommendation

Two approaches are recommended:

1. Disallow arbitrary user-defined strategy ordering. Enforce the default queue for all withdrawals so users cannot avoid specific strategies.
2. Compute the unrealized loss share across all strategies, not per-picked strategy. First aggregate an effective loss ratio over all active strategies, then deduct the user's loss pro-rata from the requested amount, regardless of the withdrawal path.

Status

Resolved

[M-04] Vault#_redeem - Withdraw-Limit Check Uses Unapplied strategies Argument (Bypass/DoS)

Description

The internal `_redeem` function validates withdrawals against the `withdraw_limit_module` using the user-supplied strategies array, but later the function may ignore that array and use the default queue instead. This mismatch lets a caller bypass policy enforced by the limit module or cause unintended reverts. The issue occurs whenever `use_default_queue` is True (or strategies is empty), causing the effective queue to differ from the one used for the limit check.

Vulnerability Details

Inside `_redeem`, the vault first checks the withdraw limit as follows, forwarding the caller's strategies argument directly to the module:

```
# Vault.vy (inside _redeem)
withdraw_limit_module: address = self.withdraw_limit_module
if withdraw_limit_module != empty(address):
                                                assert            assets            <=
IWithdrawLimitModule(withdraw_limit_module).available_withdraw_limit(
        owner, max_loss, strategies  # <-- user argument passed verbatim
    ), "exceed withdraw limit"
```

However, the function then derives the actual queue it will use to withdraw funds, which can differ from the user input:

```
# Later in _redeem
_strategies: DynArray[address, MAX_QUEUE] = self.default_queue
if len(strategies) != 0 and not self.use_default_queue:
    _strategies = strategies
```

When `use_default_queue` is True (or strategies is empty), the vault withdraws using self.default_queue, but the limit check was computed against the un-applied strategies argument. If the module's available_withdraw_limit depends on queue composition/order (e.g., per-strategy liquidity, throttles, or loss tolerances), an attacker can supply a "lenient" strategies array to pass the check, then have the vault execute against a stricter default queue—bypassing intended limits—or conversely cause erroneous reverts (DoS) when the supplied array is stricter than the default.

Impact

Withdraw throttling or policy encoded in withdraw_limit_module can be bypassed, allowing users to withdraw more assets than governance intended (e.g., exceeding per-block/day caps or ignoring per-strategy liquidity constraints). Conversely, callers can grief others by causing unnecessary reverts when the check uses a stricter, user-provided array while the vault will actually use a lenient default queue. This undermines risk controls and can accelerate liquidity drains during stress.

Recommendation

Compute the effective queue first and use that same array for both the limit check and the subsequent withdrawal logic:

```
def _redeem(..., strategies: DynArray[address, MAX_QUEUE]) -> uint256:
-    withdraw_limit_module: address = self.withdraw_limit_module
-    if withdraw_limit_module != empty(address):
-                                                        assert       assets       <=
IWithdrawLimitModule(withdraw_limit_module).available_withdraw_limit(
-            owner, max_loss, strategies
-        ), "exceed withdraw limit"
+    # Derive the effective queue once
+    _strategies: DynArray[address, MAX_QUEUE] = self.default_queue
+    if len(strategies) != 0 and not self.use_default_queue:
+        _strategies = strategies
+
+    # Enforce limits against the exact queue we will use
+    withdraw_limit_module: address = self.withdraw_limit_module
+    if withdraw_limit_module != empty(address):
+                                                        assert       assets       <=
IWithdrawLimitModule(withdraw_limit_module).available_withdraw_limit(
+            owner, max_loss, _strategies
+        ), "exceed withdraw limit"

-    # ... later recomputes _strategies
-    _strategies: DynArray[address, MAX_QUEUE] = self.default_queue
-    if len(strategies) != 0 and not self.use_default_queue:
-        _strategies = strategies
+    # ... continue with _strategies for withdrawal
```

Status

Resolved

Low

[L-01] VaultFactory#set_protocol_fee_bps - Missing Recipient Validation When Updating Default Protocol Fee

Description

The `set_protocol_fee_bps` function allows governance to update the default protocol fee without validating that a nonzero recipient address has been set. If `default_protocol_fee_config.fee_recipient` is unset, updating the fee could result in protocol fees being collected and directed to the zero address, effectively burning funds.
A similar function, `set_custom_protocol_fee_bps`, correctly performs this validation.

Recommendation

Add a recipient validation check to `set_protocol_fee_bps`, ensuring that fees cannot be configured while the recipient is the zero address. For example:

```
@external

def set_protocol_fee_bps(new_protocol_fee_bps: uint16):

    assert msg.sender == self.governance, "not governance"

    assert new_protocol_fee_bps <= MAX_FEE_BPS, "fee too high"

+       assert  self.default_protocol_fee_config.fee_recipient  !=  empty(address),  "no
recipient"


    self.default_protocol_fee_config.fee_bps = new_protocol_fee_bps


    log UpdateProtocolFeeBps(

        self.default_protocol_fee_config.fee_bps,

        new_protocol_fee_bps

    )
```

Status

Resolved

[L-02] Vault#profitUnlockingRate - Returns non-zero rate after profit window ends (stale/misleading value)

Description

`profitUnlockingRate` unconditionally returns `self.profit_unlocking_rate` even after the vault has fully unlocked profits (i.e., when `block.timestamp >= full_profit_unlock_date`). This diverges from `_unlocked_shares`, which correctly treats the rate as exhausted after the full unlock date. Off-chain consumers (UIs, analytics, bots) that rely on profitUnlockingRate() to estimate pending unlocks may display or act on stale, non-zero rates.

```
@view

@external

def profitUnlockingRate() -> uint256:

    """

    @notice The per second rate at which profits are unlocking.

    @dev This is denominated in EXTENDED_BPS decimals.

    @return The current profit unlocking rate.

    """

    return self.profit_unlocking_rate
```

Recommendation

Return 0 once profits are fully unlocked (or if no schedule exists), keeping the view in sync with `_unlocked_shares` semantics. Also clarify in natspec that the value is zeroed post-unlock.

```
@view

@external
```

```
def profitUnlockingRate() -> uint256:
```

```
1.  def profitUnlockingRate() -> uint256:
2.      """
3.      @notice The per second rate at which profits are unlocking.
4. -    @dev This is denominated in EXTENDED_BPS decimals.
5. -    @return The current profit unlocking rate.
6. +    @dev Denominated in EXTENDED_BPS decimals. Returns 0 once
   the profit
7. +          unlocking window has ended or if no schedule is active.
8. +    @return The current profit unlocking rate (0 if fully
   unlocked).
9.      """
10.   -     return self.profit_unlocking_rate
11.   +     if self.full_profit_unlock_date == 0:
12.   +         return 0
13.   +     if block.timestamp >= self.full_profit_unlock_date:
14.   +         return 0
15.   +     return self.profit_unlocking_rate
```

Status

Resolved

[L-03] Vault#setProfitMaxUnlockTime - Zero Unlock Setting Causes Arbitrary Value Shift

Description

When `setProfitMaxUnlockTime(0)` is called, the vault burns its self-held shares and resets unlocking variables, causing an immediate jump in Price-Per-Share (PPS). This discontinuity allows deposits made in the same block to benefit unfairly, since they are minted at the old PPS but can withdraw at the new higher PPS. Although this does not directly enable theft, it introduces fairness and accounting inconsistencies that can disadvantage existing LPs.

```
@external

def setProfitMaxUnlockTime(new_profit_max_unlock_time: uint256):

    self._enforce_role(msg.sender, Roles.PROFIT_UNLOCK_MANAGER)

    assert new_profit_max_unlock_time <= 31_556_952, "profit unlock time too long"



    if (new_profit_max_unlock_time == 0):

        share_balance: uint256 = self.balance_of[self]

        if share_balance > 0:

            self._burn_shares(share_balance, self)  # ↓ reduces totalSupply -> PPS jumps

        self.profit_unlocking_rate = 0

        self.full_profit_unlock_date = 0



    self.profit_max_unlock_time = new_profit_max_unlock_time

    log UpdateProfitMaxUnlockTime(new_profit_max_unlock_time)
```

Recommendation

Introduce guardrails to prevent contemporaneous deposits from capturing the discontinuity:

1. Timelock + Deposit Pause Window: queue `setProfitMaxUnlockTime(0)` with a delay and automatically pause deposits/withdrawals for `N` blocks while the change is applied.
2. Gradual Share Burn (Vesting-style unlock): instead of burning all vault-held shares immediately when `setProfitMaxUnlockTime(0)` is called, transfer them into a vesting bucket and burn them linearly over a minimum period. This ensures the PPS changes smoothly over time and prevents front-running deposits from capturing a sudden value jump.

Status

Resolved

Gas

[G-01] VaultFactory#deploy_new_vault - Reduce CREATE2 Salt Cost by Hashing Dynamic Strings

Description

`deploy_new_vault` builds the `CREATE2` salt by `ABI-encoding` two dynamic strings (name, symbol) together with `msg.sender` and asset, which forces `Vyper` to allocate/encode variable-length memory and then hash it.

```
vault_address: address = create_minimal_proxy_to(

    VAULT_ORIGINAL,

    value=0,

    salt=keccak256(_abi_encode(msg.sender, asset, name, symbol))

)
```

Since the salt only needs to be deterministic and unique, you can first hash the strings to fixed-size `bytes32` and then `ABI-encode` fixed-width values. This avoids dynamic encoding overhead and reduces gas on every deploy.

Recommendation

Pre-hash name and symbol to `bytes32` and encode only fixed-size values in the salt. This preserves uniqueness (collisions are cryptographically negligible) while cutting `memory/encoding` work.

```
@external

 def deploy_new_vault(

     asset: address,

     name: String[64],

     symbol: String[32],

     role_manager: address,
```

```
    profit_max_unlock_time: uint256
) -> address:
-   vault_address: address = create_minimal_proxy_to(
-           VAULT_ORIGINAL,
-           value=0,
-           salt=keccak256(_abi_encode(msg.sender, asset, name, symbol))
-       )
+   # Pre-hash dynamic strings to fixed-size words to avoid dynamic ABI encoding
+   name_h: bytes32 = keccak256(name)
+   symbol_h: bytes32 = keccak256(symbol)
+   vault_address: address = create_minimal_proxy_to(
+       VAULT_ORIGINAL,
+       value=0,
+       # Salt now encodes only fixed-size values (address,address,bytes32,bytes32)
+       salt=keccak256(_abi_encode(msg.sender, asset, name_h, symbol_h))
+   )
    IVault(vault_address).initialize(
        asset,
        name,
        symbol,
        role_manager,
        profit_max_unlock_time,
    )
    log NewVault(vault_address, asset)
    return vault_address
```

Status

Resolved

[G-02] Vault#DOMAIN_SEPARATOR - Recompute-on-every-call EIP-712 Domain Separator

Description

The function `_permit()` indirectly calls `domain_separator()`, which recomputes the `EIP-712` domain separator on every call. This involves repeated `keccak256` hashing and string conversions, which unnecessarily increase gas usage. A cheaper approach is to cache the domain separator once at initialization and return it directly on subsequent calls.

```
@view
@internal
def domain_separator() -> bytes32:
    return keccak256(
        concat(
            DOMAIN_TYPE_HASH,
            keccak256(convert("Balloon Vault", Bytes[13])),
            keccak256(convert(API_VERSION, Bytes[28])),
            convert(chain.id, bytes32),
            convert(self, bytes32)
        )
    )
@external
@view
def DOMAIN_SEPARATOR() -> bytes32:
    return self.domain_separator()
```

Recommendation

Cache the domain separator at `initialize` and return the stored value in `DOMAIN_SEPARATOR`. This avoids recomputation costs and ensures consistent gas efficiency.

```
-DOMAIN_SEPARATOR: computed on each call
+DOMAIN_SEPARATOR_CACHED: public(bytes32)




+@external
+def initialize(...):
+    ...
+    self.DOMAIN_SEPARATOR_CACHED = keccak256(
+        concat(
+            DOMAIN_TYPE_HASH,
+            keccak256(convert("Balloon Vault", Bytes[13])),
+            keccak256(convert(API_VERSION, Bytes[28])),
+            convert(chain.id, bytes32),
+            convert(self, bytes32)
+        )
+    )
+@external
+@view
+def DOMAIN_SEPARATOR() -> bytes32:
+    return self.DOMAIN_SEPARATOR_CACHED
```

Status

Resolved

QA

[Q-01] BalloonVaultFactory#remove_custom_protocol_fee - Emits Event on No-Op

Description

The function remove_custom_protocol_fee emits a RemovedCustomProtocolFee event even if no custom fee was previously set. This can mislead off-chain indexers and monitoring systems into assuming a state change occurred when none did.

```
self.custom_protocol_fee[vault] = 0

self.use_custom_protocol_fee[vault] = False

log RemovedCustomProtocolFee(vault)
```

Recommendation

Add a guard to check if a custom fee actually exists before proceeding.

```
- self.custom_protocol_fee[vault] = 0

- self.use_custom_protocol_fee[vault] = False

- log RemovedCustomProtocolFee(vault)

+ if self.use_custom_protocol_fee[vault]:

+     self.custom_protocol_fee[vault] = 0

+     self.use_custom_protocol_fee[vault] = False

+     log RemovedCustomProtocolFee(vault)
```

Status

Resolved

Centralisation

The Balloon project values security and utility over decentralisation.

The owner-executable functions within the protocol increase security and functionality, but depend heavily on internal team responsibility.

Conclusion

After Hashlock's analysis, the Balloon project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved. 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