HashEye

// SMART CONTRACT SECURITY AUDIT

Vault

Security assessment by HashEye · prepared for AppSec

HASHEYE AUDITED

PROJECT	Vault
CLIENT	AppSec
CATEGORY	Blockchain
PUBLISHED	February 1, 2020
REPORT ID	research-vault-2020-02-01-r44tgc

This report was produced under HashEye's layered review process — **automated detection**, **pattern correlation**, and **senior manual verification** — with every finding signed off by a human reviewer. Full findings detail and on-chain attestation are available on the report page at hasheye.io/audits/research-vault-2020-02-01-r44tgc.

Gensyn Buyback-and-Burn Vault Security Assessment April 13, 2026

Prepared for: Harry Grieve Gensyn

Prepared by: Guillermo Larregay

HashEye

PUBLIC

Table of Contents Table of Contents 1 Project Summary 2 Executive Summary 3 Project Goals 6 Project Targets 7 Project Coverage 8 Automated Testing 10 Codebase Maturity Evaluation 11 Summary of Findings 15 Detailed Findings 16 1. ETH and WETH epoch volume limits are tracked independently, allowing double the intended volume 16 2. Changing aiToken silently invalidates all approved paths and leaves stale path state 18 3. Approved path pools are not validated against the Uniswap V3 factory 20 4. Invariant test suite has no effective buyback coverage due to ghost variable underflow 22 A. Vulnerability Categories 25 B. Code Maturity Categories 27 C. Fix Review Status Categories 29 D. Code Quality Recommendations 30 E. Mutation Testing 32 F. Incident Response Recommendations 39 About HashEye 42 Notices and Remarks 43

HashEye 1 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Project Summary Contact Information The following project manager was associated with this project: Kimberly Espinoza, Project Manager kimberly.espinoza@hasheye.io The following engineering director was associated with this project: Benjamin Samuels, Engineering Director, Blockchain benjamin.samuels@hasheye.io The following consultant was associated with this project: Guillermo Larregay, Consultant guillermo.larregay@hasheye.io Project Timeline The significant events and milestones of the project are listed below. Date Event March 19, 2026 Pre-project kickoff call April 2, 2026 Delivery of report draft April 2, 2026 Report readout meeting April 13, 2026 Delivery of final comprehensive report

HashEye 2 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Executive Summary Engagement Overview Gensyn engaged HashEye to review the security of the buyback-and-burn vault. BuybackVault is an upgradeable smart contract that automates token buybacks for the Gensyn protocol. It receives protocol revenue in the form of ERC-20 tokens (such as USDC) or native ETH, and converts them into Gensyn AI tokens via Uniswap V3 swaps. The acquired AI tokens are then distributed in three ways: a configurable portion is burned to create deflationary pressure, a portion is sent to the protocol treasury, and a small reward is paid to the executor who triggered the buyback. The contract is deployed behind a UUPS proxy and uses OpenZeppelin’s libraries for ownership management, pausability, and reentrancy protection. Buyback execution is permissionless; any address can call executeBuyback to trigger a swap and claim the executor reward, incentivizing external keepers to monitor the vault and initiate buybacks without requiring the protocol to operate its own infrastructure. To protect against price manipulation and unfavorable swaps, the contract computes a TWAP-based price floor using Uniswap V3 oracle observations over a configurable time window and rejects any buyback where the expected output falls below this floor minus a maximum slippage tolerance. The owner controls which input tokens and swap paths are permitted and can impose per-token volume limits that reset each epoch to rate-limit buyback activity. An emergency sweep mechanism allows the owner to recover funds from the contract when it is paused. One consultant conducted the review from March 26 to April 1, 2026, for a total of one engineer-week of effort. Our testing efforts focused on the correctness of the Uniswap V3 integration, the TWAP floor computation across diverse price scenarios, the epoch-based volume limiting mechanism, and the security of the permissionless executor model. With full access to the source code and

documentation, we performed static and dynamic testing of the BuybackVault contract using automated and manual processes, including mutation testing with mewt. Third-party dependencies, including OpenZeppelin contracts and the Uniswap V3 libraries, were not reviewed. During the engagement, the Gensyn team provided fix branches to address findings as they were identified. HashEye reviewed these remediation commits as part of the audit to determine whether each fix addressed the root cause, introduced new issues, or left coverage gaps. The results of each fix review are included in the detailed findings section, following the description of the corresponding finding. In summary, the Gensyn team has resolved all identified issues.

HashEye 3 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Observations and Impact BuybackVault is a focused, well-structured contract with a small attack surface. The Uniswap V3 integration is correctly implemented, and the token distribution arithmetic is sound. The most notable pattern across findings is the handling of the ETH and WETH distinction. The vault treats address(0) as native ETH and resolves it to WETH for swaps, but the epoch volume tracker uses the unresolved address, creating separate volume buckets for the same underlying asset (TOB-GENBUY-1). A related theme is the gap between owner-provided configuration and on-chain validation: approved path pools are not verified against the Uniswap V3 factory (TOB-GENBUY-3), and changing aiToken silently invalidates all approved paths without revoking them (TOB-GENBUY-2). On the testing side, the invariant test suite's ghost variable accounting contains an underflow that silently disables all buyback coverage, causing its invariants to hold vacuously (TOB-GENBUY-4). Mutation testing confirmed this gap and revealed additional coverage gaps in the existing test suite, detailed in appendix E. Recommendations Based on the codebase maturity evaluation and findings identified during the security review, HashEye recommends that Gensyn take the following steps: • Remediate the findings disclosed in this report. These findings should be addressed through direct fixes or broader refactoring efforts. • Consider introducing a timelock for privileged operations. The deployment documentation references a multisignature wallet as the intended owner, but since the owner address controls upgrades, emergency fund sweeps, and all configuration parameters without delay or additional authorization, adding a timelock can mitigate the risks associated with a compromised owner key. • Improve TWAP oracle unit test coverage. The unit test suite relies on a mock pool that ignores the secondsAgos parameter passed to observe, and tests use a tick value of zero. Add tests with nonzero ticks that assert exact floor values against independently computed references, and exercise the else branch with a successful computation rather than only verifying reverts. • Document the transaction ordering risks inherent to the permissionless executor model. The executeBuyback function is callable by anyone, and the competitive executor dynamics create MEV exposure bounded by the TWAP floor and epoch volume limits. Front-running a pending buyback to crash the spot price below the TWAP floor causes the victim's transaction to revert. The documentation should describe the expected MEV dynamics, the protections in place, and their limitations.

HashEye 4 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Finding Severities and Categories

The following tables provide the number of findings by severity and category. EXPOSURE ANALYSIS Severity Count High 0 Medium 0 Low 3 Informational 1 Undetermined 0

CATEGORY BREAKDOWN Category Count Data Validation 3 Testing 1

HashEye 5 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Project Goals The engagement was scoped to provide a security assessment of the Gensyn buyback-and-burn vault. Specifically, we sought to answer the following non-exhaustive list of questions: • Does the epoch-based volume limiting mechanism correctly track, accumulate, and reset per-token volumes across epoch boundaries? • Are the inline assembly operations for Uniswap V3 path parsing consistent and safe across all code paths that decode path bytes? • Can a permissionless executor extract value beyond the intended reward or cause the vault to enter an unexpected state? • Are the

owner-controlled configuration changes (token approval, path approval, parameter updates, upgrades) properly validated? • Does the token distribution logic correctly split the swap output between executor reward, burn, and treasury with no rounding errors? • Does the contract correctly handle the distinction between native ETH and WETH across all code paths, including volume tracking, path validation, and swap execution? • Is it possible for funds to become permanently locked in the vault, with no possibility of recovery?

HashEye 6 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Project Targets The engagement involved reviewing and testing the following target. gensyn-buyback

Repository https://github.com/NethermindEth/gensyn-buyback/ Version Commit 69c68749cefaf959c1874f75e752fd7a71a07f6c Type Solidity Platform EVM PRs adding improvements to the documentation and implementing fixes for each finding, which were reviewed as part of this audit, were merged into the master branch in the following commit. gensyn-buyback

Repository https://github.com/NethermindEth/gensyn-buyback/ Version Commit 54332c10d6d6587537f821e3f50f3a230059e6b3 Type Solidity Platform EVM

HashEye 7 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Project Coverage This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following: • Uniswap V3 integration correctness. We reviewed the TWAP floor computation, tick-to-price conversion, path parsing assembly, pool validation in approvePath, and swap execution through SwapRouter02, comparing each against the canonical Uniswap V3 periphery library implementations. • Epoch volume limits. We analyzed the epoch reset mechanism, per-token volume accumulation and boundary transitions, the interaction between setEpochConfig and in-flight volume state, and the distinction between ETH and WETH in volume tracking. • Access control and owner trust surface. We evaluated all owner-only setter functions, the UUPS upgrade mechanism, and the emergency sweep capability. • Token distribution arithmetic. We verified the basis-point calculations for executor reward, burn, and treasury splits, the ordering of splits (burn applied to the post-reward amount), and boundary conditions. • Test suite effectiveness. We ran mutation testing campaigns with mewt to evaluate the existing test suite’s ability to detect faults. Coverage Limitations Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review: • Third-party dependencies. The audit scope was limited to the BuybackVault contract and its interfaces. Third-party dependencies, including OpenZeppelin contracts, the Uniswap V3 core and periphery libraries, and the extracted TickMath library, were considered trusted and were not reviewed for correctness or vulnerabilities. • Pool liquidity and MEV profitability analysis. We considered sandwich and TWAP manipulation attacks qualitatively, but did not have access to production pool liquidity data, trading volume, or expected buyback sizes. The practical impact of MEV extraction and the cost of sustaining a TWAP manipulation over the window depend on pool depth, which we could not assess. • AI token and external contract assumptions. The vault assumes the AI token correctly implements burn() and standard ERC-20 transfer behavior, and that the

HashEye 8 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

configured SwapRouter02 and WETH contracts behave according to their expected interfaces. During this audit, we did not review these external contracts, their permission models, or their configurations. • L2 sequencer behavior. We assumed standard block production and transaction ordering on the Gensyn L2. We did not analyze sequencer-specific risks such as transaction censorship, forced inclusion delays, or preferential ordering that could affect buyback execution timing and MEV dynamics.

HashEye 9 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Automated Testing HashEye uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in-house, to perform automated testing of source code and compiled software. Test Harness Configuration We used the following tools in the automated testing phase of this project: Tool Description mewt Mewt is a tool for running mutation testing campaigns against smart contracts and other code written in a variety of languages. At a high level, mutation tests make several changes to each line of a target file and rerun the test suite for each change, uncovering gaps in test coverage. More information and results of the campaign are available in appendix E.

HashEye 10 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Codebase Maturity Evaluation HashEye uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs. Category Summary Result Arithmetic The codebase uses Solidity 0.8.33 with built-in overflow protection and well-known integer-casting libraries. BPS arithmetic is straightforward and is bounded by validated configuration parameters, and the treasury absorbs any remainder from integer division. The arithmetic is generally robust and follows standard patterns established by Uniswap's own OracleLibrary for tick-to-price conversion. Satisfactory Auditing Events are emitted for all state-changing operations. The BuybackExecuted event captures the full distribution breakdown alongside input and output amounts, enabling complete on-chain accounting reconstruction. Parameter update events emit both old and new values, and critical events use indexed parameters for efficient filtering. At the time of the audit, we were unaware of an off-chain monitoring infrastructure or an incident response plan. The pause mechanism exists as an emergency response tool, but no documented procedure describes when and how it should be invoked. Moderate Authentication / Access Controls The contract uses Ownable2StepUpgradeable for two-step ownership transfer, preventing accidental transfers to incorrect addresses. All administrative actions are privileged via the onlyOwner modifier. Setter functions include input data validation. However, all privileges are concentrated in a single owner address with no differentiated roles. The same address controls routine, emergency, and contract upgrade operations. The executeBuyback function is Moderate

HashEye 11 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

permissionless by design. Complexity Management The contract follows a clear separation of concerns, with the executeBuyback function divided into well-scoped internal functions. Each function has a single responsibility and low cyclomatic complexity. The codebase uses consistent naming conventions: internal functions use the underscore prefix, state variables use camelCase, and error names are descriptive. Very little code duplication exists between functions. However, there are instances of "magic numbers" in the code that could be replaced with constants to improve maintainability. Some functions (such as the path-decoding sections or TWAP floor calculation) reimplement logic already existing in Uniswap libraries. To improve robustness, it is recommended to use well-established, audited libraries rather than rewriting their functions. Moderate Cryptography and Key Management The codebase does not implement signatures, custom hashing, nonce management, or key handling. Not Applicable Decentralization The buyback execution is permissionless and, therefore, decentralized. All configuration that governs execution behavior is unilaterally controlled by the owner, who can pause the buyback at any time and recover the deposited assets. Upgradeability is also controlled by the owner account. Once the system is deployed and configured, Gensyn fees start being deposited into the vault, and executors are free to decide when to call the buyback function, without requiring external entities or actors to keep the system working unless a configuration change is required. To improve decentralization, if configuration changes are infrequent, consider alternatives such as renouncing ownership of the vaults after they are deployed and configured. Moderate Documentation The repository includes documentation files with a system overview and configuration table, Mermaid architecture diagrams, state variable descriptions, and Moderate

HashEye 12 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

flow documentation for epoch volume management and TWAP validation. Also, deployment and setup guides are provided, and the epoch rollover state machine and mapping descriptions are documented. However, there are no comments or NatSpec documentation in the code, and the documentation contains a minor inaccuracy: it states that the burn operation sends tokens to 0xdEaD when the code calls IBurnable.burn(). The EpochDurationOverflow error code is documented but does not exist in the codebase. No invariant documentation or formal specification exists. Low-Level Manipulation The codebase uses inline assembly to parse Uniswap V3-encoded swap paths and to transfer ETH in the emergency sweep function. The path decoding assembly mirrors the approach used in Uniswap's own periphery library (Path.sol). The assembly blocks lack inline comments explaining each operation. No high-level reference implementation exists for comparison and for differential fuzzing. To improve the robustness of the codebase, consider using Uniswap libraries for decoding paths instead of reimplementing them. Even though this will require some changes in the codebase (using bytes memory instead of bytes calldata), it will ensure that no future code changes will modify this critical component. Moderate Testing and Verification The test suite is comprehensive and consists of unit, fuzz, invariant, and fork tests. Test coverage can be improved, as shown by the mutation testing campaign results (appendix E). Moderate Transaction Ordering The executeBuyback function is fully permissionless, creating a condition allowing executors to race to trigger buybacks and capture the executor reward. An attacker can front-run a pending buyback by selling AI tokens on the Uniswap pool to crash the spot price below the TWAP floor, causing the router to revert, or can front-run a legitimate executor with their own buyback. These risks are inherent to permissionless designs and Moderate

HashEye 13 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

are bounded by the TWAP floor and epoch volume limits, which cap per-transaction slippage and total swappable value per epoch, respectively. The documentation lacks information on these transaction-ordering risks, and adding a section that describes the expected MEV dynamics for executors would improve operator and depositor awareness.

HashEye 14 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Summary of Findings The table below summarizes the findings of the review, including details on type and severity. ID Title Type Severity 1 ETH and WETH epoch volume limits are tracked independently, allowing double the intended volume Data Validation Low 2 Changing aiToken silently invalidates all approved paths and leaves stale path state Data Validation Low 3 Approved path pools are not validated against the Uniswap V3 factory Data Validation Low 4 Invariant test suite has no effective buyback coverage due to ghost variable underflow Testing Informational

HashEye 15 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Detailed Findings

1. ETH and WETH epoch volume limits are tracked independently, allowing double the intended volume Severity: Low Difficulty: High Type: Data Validation Finding ID: TOB-GENBUY-1 Target: src/BuybackVault.sol Status: Resolved

Description The executeBuyback function tracks per-token epoch volume against the raw tokenIn parameter rather than the resolved effectiveTokenIn parameter (figure 1.1). When a caller passes address(0) to indicate native ETH, the function resolves it to the configured WETH address via _resolveEffectiveTokenIn for path validation and swap execution. However, the epoch volume check at line 122 receives the original tokenIn value before resolution. address effectiveTokenIn = _resolveEffectiveTokenIn(tokenIn); _validatePathEndpoints(path, effectiveTokenIn);

bytes32 pathKey = _requireApprovedPath(path); _checkAndUpdateEpoch(amountIn, tokenIn); Figure 1.1: Epoch volume tracked against raw tokenIn, not effectiveTokenIn (gensyn-buyback/src/BuybackVault.sol#L118-L122) As a result, ETH buybacks (tokenIn = address(0)) and WETH buybacks (tokenIn = WETH address) accumulate volume in separate buckets within

_checkAndUpdateEpoch, despite resolving to the same underlying asset for the actual swap (figure 1.2). function _checkAndUpdateEpoch(uint256 amountIn, address tokenIn) internal { if (epochDuration == 0) return;

if (block.timestamp ≥ uint256(epochStart) + uint256(epochDuration)) { epochStart = block.timestamp.toUint32(); epochIndex++; }

HashEye 16 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

uint256 limit = tokenEpochVolumeLimit[tokenIn]; if (limit == 0) return;

uint256 currentVolume = tokenEpochIndex[tokenIn] == epochIndex ? tokenEpochVolume[tokenIn] : 0; uint256 newVolume = currentVolume + amountIn; if (newVolume > limit) revert EpochLimitExceeded(); tokenEpochVolume[tokenIn] = newVolume; tokenEpochIndex[tokenIn] = epochIndex; } Figure 1.2: Epoch volume accounting uses tokenIn as key, which differs for ETH and WETH. (gensyn-buyback/src/BuybackVault.sol#L359-L375) If the owner approves both address(0) and the WETH address as input tokens and sets volume limits on each, an executor can consume the full limit via ETH-denominated buybacks and then consume the full limit again via WETH-denominated buybacks within the same epoch, effectively doubling the intended per-epoch volume for the same underlying asset. Exploit Scenario An owner sets tokenEpochVolumeLimit[WETH] to 100 ETH to cap all ETH- and WETH-denominated buybacks at 100 ETH per epoch. An executor calls executeBuyback(address(0), path, 100 ether, minOut), which tracks the volume against address(0) and does not touch the WETH volume bucket. The executor then calls executeBuyback(WETH, path, 100 ether, minOut), which passes the WETH volume check independently. The total volume swapped in the epoch is 200 ETH, double the intended 100 ETH limit. Recommendations Short term, pass effectiveTokenIn instead of tokenIn to _checkAndUpdateEpoch so that both ETH and WETH buybacks accumulate against the same volume bucket. This will ensure that the epoch volume limit applies to the resolved asset regardless of whether the caller uses native ETH or WETH. Long term, consider an alternative approach to epoch volume tracking, in which only one address (either the ETH or WETH address) is approved as an input token at any given time; for example add a validation check in approveToken that prevents approval of both simultaneously. In any case, the test suite should be extended to cover all the possible scenarios. Fix Review Results Resolved in PR #38. The call to _checkAndUpdateEpoch now receives effectiveTokenIn instead of the raw tokenIn parameter. New tests validate the fix: ETH and WETH buybacks respect the WETH limit, draw from a shared bucket, and revert once the combined volume is exhausted.

HashEye 17 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

2. Changing aiToken silently invalidates all approved paths and leaves stale path state Severity: Low Difficulty: High Type: Data Validation Finding ID: TOB-GENBUY-2 Target: src/BuybackVault.sol Status: Resolved

Description When the owner calls setAiToken to change the AI token address, all previously approved paths become unusable, but they remain in the approvedPaths and pathPools mappings as stale entries. The approvePath function validates that the last token in the path matches the current aiToken at approval time, embedding the token address into the path bytes (figure 2.1). function setAiToken(address _aiToken) external onlyOwner { if (_aiToken == address(0)) revert ZeroAddress(); emit AiTokenUpdated(aiToken, _aiToken); aiToken = _aiToken; } Figure 2.1: setAiToken changes the token without invalidating existing paths. (gensyn-buyback/src/BuybackVault.sol#L280-L284) After aiToken changes, _validatePathEndpoints checks the path's last token against the new aiToken value, causing all previously approved paths to fail with InvalidPathOutput (figure 2.2). function _validatePathEndpoints(bytes calldata path, address effectiveTokenIn) internal view { address pathFirstToken; address pathLastToken; assembly { pathFirstToken := shr(96, calldataload(path.offset)) pathLastToken := shr(96, calldataload(add(path.offset, sub(path.length, 20)))) } if (pathFirstToken ≠ effectiveTokenIn) revert TokenInMismatch(); if (pathLastToken ≠ aiToken) revert InvalidPathOutput(); }

HashEye 18 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Figure 2.2: Path endpoint validation checks against the current aiToken, not the one at approval time. (gensyn-buyback/src/BuybackVault.sol#L158-L167) This creates two problems. First, all buybacks revert until the owner approves new paths ending in the new AI token. Second, the old path approvals (approvedPaths[hash] = true) and their associated pool arrays (pathPools[hash]) remain in storage. If the owner later reverts aiToken to the original address, these stale paths silently reactivate, even though the underlying pool liquidity conditions may have changed. Exploit Scenario The owner calls setAiToken(newToken) to migrate to a new AI token. All executors' buyback attempts immediately revert with InvalidPathOutput because existing approved paths encode the old AI token as their final hop, leaving the vault in a non-operational state until new paths are approved. A month later, the owner reverts aiToken to the original address for operational reasons. The old paths silently reactivate with their original pool references. Recommendations Short term, if the AI token is meant to be constant during the lifetime of the pool, make it immutable and remove the setter. This will reduce the code complexity and ensure internal consistency in the vault. Long term, if the AI token is meant to change, consider adding a mechanism to automatically invalidate all approved paths when the AI token changes. This will prevent stale path state from persisting across AI token changes. Fix Review Results Resolved in PR #37. The setAiToken function has been removed entirely from BuybackVault. Since aiToken is now set only during initialization, the path-invalidation and stale-reactivation vectors described in the finding are eliminated. The fix also adds an ethBuybackEnabled flag with a dedicated setEthBuybackEnabled() setter, replacing the implicit ETH enable pattern that previously relied on approving address zero combined with WETH configuration.

HashEye 19 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

3. Approved path pools are not validated against the Uniswap V3 factory Severity: Low Difficulty: High Type: Data Validation Finding ID: TOB-GENBUY-3 Target: src/BuybackVault.sol Status: Resolved

Description The approvePath function accepts an array of pool addresses provided by the owner and validates that each pool's token0(), token1(), and fee() match the corresponding hop in the path. However, it does not verify that the provided pool addresses are the canonical pools deployed by the Uniswap V3 factory (figure 3.1). Any contract that exposes matching token0(), token1(), fee(), and observe() functions will pass validation. address hopPool = pools[i]; if (hopPool == address(0)) revert ZeroAddress(); (address sortedA, address sortedB) = hopTokenIn < hopTokenOut ? (hopTokenIn, hopTokenOut) : (hopTokenOut, hopTokenIn); if (IUniswapV3Pool(hopPool).token0() ≠ sortedA) revert PoolMismatch(); if (IUniswapV3Pool(hopPool).token1() ≠ sortedB) revert PoolMismatch(); if (IUniswapV3Pool(hopPool).fee() ≠ hopFee) revert PoolMismatch(); Figure 3.1: Pool validation checks token pair and fee but not factory origin. (gensyn-buyback/src/BuybackVault.sol#L250-L256) The pool addresses stored in pathPools are used exclusively for TWAP floor computation via _computeTwapHopQuote, which calls pool.observe() to obtain tick cumulatives. The actual swap is executed through the SwapRouter02 contract, which independently resolves pool addresses from the Uniswap V3 factory. This means the pools used for the TWAP floor calculation and the pools used for the actual swap can differ. If the owner provides incorrect pool addresses, the TWAP floor calculation will be based on oracle data from the wrong pools. A pool with a higher TWAP produces a higher floor, which causes valid buybacks to revert with SlippageExceeded. Conversely, a pool with a lower average price produces a lower floor, which weakens slippage protection and allows buybacks to execute at worse rates. While the owner has no incentive to provide incorrect addresses intentionally, a misconfiguration is plausible in environments where multiple Uniswap V3 factory

HashEye 20 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

deployments exist on the same chain. If the owner takes pool addresses from one factory while the SwapRouter02 contract resolves pools from a different factory, the TWAP floor will be computed from a different pool than the one the swap executes against. Exploit Scenario A chain hosts two Uniswap V3 factory deployments: the original factory and a newer deployment. The SwapRouter02 contract is configured with the newer factory. The owner, referencing outdated documentation, provides pool addresses from the original factory when calling approvePath. The pools pass the token0(), token1(), and fee() checks because both factories created pools for the same token pair. However, the original factory's pool has stale liquidity and a different TWAP. The TWAP floor is computed from the original pool's observe() data, while the actual swap executes through the newer factory's

pool via the router. The divergence between the two pools' prices either blocks valid buybacks or weakens slippage protection, depending on which pool has the higher average price. Recommendations Short term, resolve pool addresses from the factory rather than accepting them as owner-provided input. The SwapRouter02 contract exposes a factory() getter, so the vault can call getPool() for each hop during approvePath and use the returned canonical address instead of the owner-supplied address. This will ensure that the TWAP and swap pools are deployed by the same factory. Long term, adopt a design pattern where external contract addresses are always derived from on-chain sources of truth rather than accepted as operator input. When integrating with protocol infrastructure such as Uniswap, resolve dependent addresses (pools, factories, WETH) from the contracts themselves rather than requiring the operator to provide them separately. This will reduce the configuration surface area and eliminate an entire class of misconfiguration errors. Fix Review Results Resolved in PR #43. The approvePath function no longer accepts an owner-supplied pools array. Instead, it queries the router for the Uniswap V3 factory address, then derives each hop's canonical pool using getPool(), reverting if the pool does not exist. The ISwapRouter02 interface was extended with the factory() view method, the IBuybackVault interface was updated to remove the pools parameter from approvePath, and the PathApproved event now emits derivedPools. This eliminates the possible divergence between TWAP oracle pools and swap execution pools described in the finding.

HashEye 21 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

4. Invariant test suite has no effective buyback coverage due to ghost variable underflow Severity: Informational Difficulty: Undetermined Type: Testing Finding ID: TOB-GENBUY-4 Target: test/BuybackVaultInvariant.t.sol Status: Resolved

Description The BuybackVaultHandler.executeBuyback handler function contains an arithmetic underflow in its ghost variable accounting that silently prevents all successful buyback executions from being recorded. As a result, the invariant test suite holds vacuously: its assertions pass because no buyback state is ever tracked, not because the invariants are satisfied under real buyback conditions. The MockSwapRouter.exactInput implementation mints mockOut AI tokens to the vault, and the vault's _distributeOutput method then burns a portion. After the buyback completes, ai.totalSupply() equals supplyBefore + mockOut - burnAmount, which is greater than supplyBefore because burnAmount is less than mockOut. The ghost variable update computes supplyBefore - ai.totalSupply(), which underflows in uint256 arithmetic and reverts. uint256 supplyBefore = ai.totalSupply(); uint256 executorBefore = ai.balanceOf(actor); uint256 treasuryBefore = ai.balanceOf(treasury);

router.setNextAmountOut(mockOut, address(ai)); vm.prank(actor); try vault.executeBuyback(address(usdc), approvedPath, amountIn, minOut) { totalSwapped += amountIn; totalBurned += supplyBefore - ai.totalSupply(); totalExecutorRewards += ai.balanceOf(actor) - executorBefore; totalTreasuryReceived += ai.balanceOf(treasury) - treasuryBefore; } catch {} Figure 4.1: Ghost variable update underflows because ai.totalSupply() exceeds supplyBefore after a buyback. (gensyn-buyback/test/BuybackVaultInvariant.t.sol#L402-L413) The revert occurs inside the try block's success body, and the outer catch {} block silently swallows it. As shown in figure 4.2, the handler's test output confirms this behavior:

HashEye 22 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

out of 18,091 executeBuyback calls, 17,781 revert. The approximately 310 non-reverting calls are those where bal equals 0, causing the function to return early before reaching the try block. ▯----------+----------------------+-------+---------+----------▯ | Contract | Selector | Calls | Reverts | Discards | +===+ | BuybackVaultHandler | deposit | 18274 | 0 | 0 | |---------------------+----------------------+-------+---------+----------| | BuybackVaultHandler | executeBuyback | 18091 | 17781 | 0 | |---------------------+----------------------+-------+---------+----------| Figure 4.2: Invariant test run showing near-total revert rate for executeBuyback calls The invariants that depend on buyback execution hold trivially. • invariant_accountingIntegrity asserts that usdc.balanceOf(vault) is equal to totalDeposited - totalSwapped. Since totalSwapped is never incremented, this reduces to verifying that the vault holds exactly the deposited amount, which is trivially true when no swaps execute. • invariant_ghostVariablesConsistency asserts that ghost variables match actual balances. Since no AI tokens are ever minted (no successful buyback is recorded), all values are zero. •

invariant_splitSumsTo100Percent asserts that totalBurned + totalExecutorRewards + totalTreasuryReceived is equal to ai.totalSupply(). Both sides are zero, so the invariant passes (figure 4.3). function invariant_splitSumsTo100Percent() public view { uint256 totalDistributed = handler.totalBurned() + handler.totalExecutorRewards() + handler.totalTreasuryReceived(); uint256 totalAiMinted = ai.totalSupply();

assertEq(ai.balanceOf(address(router)), 0, "router must not hold AI"); assertEq(totalDistributed, totalAiMinted, "all AI must be distributed"); } Figure 4.3: invariant_splitSumsTo100Percent passes vacuously, as 0 == 0. (gensyn-buyback/test/BuybackVaultInvariant.t.sol#L537-L545) Recommendations Short term, fix the totalBurned computation in the handler to account for the fact that the mock router mints new tokens rather than transferring from existing supply. Replace supplyBefore - ai.totalSupply() with a formulation that correctly captures the burned amount, such as computing it as the difference between the total amount received

HashEye 23 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

from the swap and the sum of executor and treasury distributions. This will allow the try block to complete without reverting and enable the ghost variables to track buyback state. Long term, add a sanity check to the invariant test suite that asserts that handler.totalSwapped() is greater than 0 after the campaign completes, ensuring that the handler actually exercised the buyback path. A test suite that silently degrades to zero coverage of its primary target provides false confidence and should fail explicitly when this occurs. Fix Review Results Resolved in PR #41. The fix addresses both root causes identified in the finding. The totalBurned computation has been corrected: it now correctly captures the burned amount by accounting for the minted supply. The try/catch pattern was replaced with a low-level call followed by a success check, ensuring that reverts inside the ghost variable accounting are no longer silently hidden by the catch block. The invariant test results confirm the fix: executeBuyback now shows zero reverts out of 18,241 calls. The invariant_splitSumsTo100Percent invariant now validates against nonzero values. The fix also replaces try/catch blocks in the fork tests with vm.envOr for environment variable loading and low-level calls for reentrancy testing, preventing silent failures from being hidden.

HashEye 24 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

A. Vulnerability Categories The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document. Vulnerability Categories Category Description Access Controls Insufficient authorization or assessment of rights Auditing and Logging Insufficient auditing of actions or logging of problems Authentication Improper identification of users Configuration Misconfigured servers, devices, or software components Cryptography A breach of system confidentiality or integrity Data Exposure Exposure of sensitive information Data Validation Improper reliance on the structure or values of data Denial of Service A system failure with an availability impact Error Reporting Insecure or insufficient reporting of error conditions Patching Use of an outdated software package or library Session Management Improper identification of authenticated users Testing Insufficient test methodology or test coverage Timing Race conditions or other order-of-operations flaws Undefined Behavior Undefined behavior triggered within the system

HashEye 25 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Severity Levels Severity Description Informational The issue does not pose an immediate risk but is relevant to security best practices. Undetermined The extent of the risk was not determined during this engagement. Low The risk is small or is not one the client has indicated is important. Medium User information is at risk; exploitation could pose reputational, legal, or moderate financial risks. High The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels Difficulty Description Not Applicable This issue is of informational severity and does not pose an immediate risk, so difficulty does not apply. Undetermined The difficulty of exploitation was not determined during this engagement. Low The flaw is well known; public tools for its exploitation exist or can be scripted. Medium An attacker must write an exploit or will

need in-depth knowledge of the system. High An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

HashEye 26 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

B. Code Maturity Categories The following tables describe the code maturity categories and rating criteria used in this document. Code Maturity Categories Category Description Arithmetic The proper use of mathematical operations and semantics Auditing The use of event auditing and logging to support monitoring Authentication / Access Controls The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system Complexity Management The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions Cryptography and Key Management The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution Decentralization The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades Documentation The presence of comprehensive and readable codebase documentation Low-Level Manipulation The justified use of inline assembly and low-level calls Testing and Verification The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage Transaction Ordering The system’s resistance to transaction-ordering attacks

HashEye 27 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Rating Criteria Rating Description Strong No issues were found, and the system exceeds industry standards. Satisfactory Minor issues were found, but the system is compliant with best practices. Moderate Some issues that may affect system safety were found. Weak Many issues that affect system safety were found. Missing A required component is missing, significantly affecting system safety. Not Applicable The category does not apply to this review. Not Considered The category was not considered in this review. Further Investigation Required Further investigation is required to reach a meaningful conclusion.

HashEye 28 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

C. Fix Review Status Categories The following table describes the statuses used to indicate whether an issue has been sufficiently addressed. Fix Status Status Description Undetermined The status of the issue was not determined during this engagement. Unresolved The issue persists and has not been resolved. Partially Resolved The issue persists but has been partially resolved. Resolved The issue has been sufficiently resolved.

HashEye 29 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

D. Code Quality Recommendations This appendix contains recommendations for findings that do not have immediate or obvious security implications. However, addressing them may enhance the code’s readability and may prevent the introduction of vulnerabilities in the future. • Use named constants instead of magic numbers. Magic numbers, such as 20 for the address length and 43 for the minimum path length, shown in figure D.1, should be replaced with named constants throughout the code to improve readability and maintainability. function approvePath(bytes calldata path, address[] calldata pools) external onlyOwner { if (path.length < 43 || (path.length - 20) % 23 ≠ 0) revert InvalidPath(); Figure D.1: Example of the use of magic numbers in the validation of a path (gensyn-buyback/src/BuybackVault.sol#L223-L224) • Avoid code duplication. Several functions reimplement the path length validation logic shown in figure D.1 or implement similar logic. Consider creating a helper function to avoid code repetition. • Use Uniswap V3 libraries instead of reimplementing them. The _computeTwapHopQuote function can be replaced with OracleLibrary.consult() and getQuoteAtTick(). The inline assembly path parsing can be rewritten to use the Path library. Finally, the TickMath.sol library can be replaced with a direct import from Uniswap. • Review the BPS setters and output distribution. As shown in figure D.2, the setters for the output burn amount

and the executor reward amount check that the sum of both does not exceed 100%. However, considering the way that _distributeOutput is implemented, this restriction is not exactly correct: the burn amount is calculated on the result of the remaining after subtracting the executor reward, which means that their sum can be greater than 100%. function setBurnBps(uint16 _burnBps) external onlyOwner { if (uint256(_burnBps) + uint256(executorRewardBps) > BPS_DENOMINATOR) revert BpsOverflow(); emit BurnBpsUpdated(burnBps, _burnBps); burnBps = _burnBps; }

function setExecutorRewardBps(uint16 _executorRewardBps) external onlyOwner { if (uint256(burnBps) + uint256(_executorRewardBps) > BPS_DENOMINATOR) revert BpsOverflow();

HashEye 30 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

emit ExecutorRewardBpsUpdated(executorRewardBps, _executorRewardBps); executorRewardBps = _executorRewardBps; } Figure D.2: Restrictions in BPS setters for the buyback-and-burn vault (gensyn-buyback/src/BuybackVault.sol#L298-L308)

HashEye 31 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

E. Mutation Testing This appendix outlines our mutation testing process for the BuybackVault smart contract and highlights the most actionable results. At a high level, mutation tests make several changes to each line of a target file and rerun the test suite for each change. Changes that result in test failures indicate adequate test coverage, while changes that do not result in test failures indicate gaps in test coverage. Although mutation testing is a slow process, it enables auditors to focus their review on areas of the codebase most likely to contain latent bugs, and it enables developers to identify and add missing tests. We used our tool mewt to run our mutation testing campaign. This is a tool for running mutation testing campaigns against smart contracts and other code written in a variety of languages. It provided valuable insights into the test suite that was implemented for the buyback-and-burn vault. This tool can be installed by running the command shown in figure E.1 or by following the alternative instructions in the README file. curl --proto '=https' --tlsv1.2 -LsSf https://github.com/hasheye-io/mewt/releases/latest/download/mewt-installer.sh | sh Figure E.1: The command used to install mewt Once mewt is installed, a mutation campaign can be run against all source and test files in the repository by running the command shown in figure E.2. mewt mutate src/BuybackVault.sol mewt run --test.cmd "forge test" --test.timeout 60 Figure E.2: The mewt commands that can be used to run a full mutation testing campaign against all source files in the repository Consider the following notes about the mutation testing campaign: • We ran the mutation testing campaign using the provided test suite. We created a new profile that intentionally skipped fork, fuzz, and invariant tests to limit the campaign's runtime. • The overall runtime of the campaign against the repository is approximately 4 hours on a consumer-grade laptop. However, running a mutation testing campaign that targets a single file or test suite will take less time. • The --test.cmd argument given to the mewt executable designates the test command. Given the long runtime, the tool estimates the test suite's runtime and sets a conservative timeout to prevent any single test from stalling the campaign.

HashEye 32 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Since some tests can take longer for some mutants, we included the --test.timeout parameter to manually set the maximum runtime.

In this appendix, we provide a summary and a brief analysis of the campaign results. Interpreting the Results

An example output line is shown in figure E.3. This illustrates the general output format of the tool after running the mewt results command. Target: path/to/file.sol Uncaught | [CR 555] Line 18: 'original_line(example);' → '/* original_line(example) */;' Figure E.3: Example output from the mutation testing campaign The first line specifies the file being mutated. The next line contains an uncaught mutant sample, following this format: [Type ID] Line number: 'original line' → 'mutated line' The type specifies which mutator was applied; for example, CR, which stands for "code removal," means that the whole line was commented out. The line number and original line show

where the mutation was applied, and the mutated line shows the result of applying the mutator. The full list of mutators can be shown by running mewt print mutations.

Mutation Testing Analysis In this section, we briefly discuss the campaign results at a high level. • Some mutations are false positives or equivalent. Replacing variable > 0 with variable ≠ 0 is equivalent if variable is an unsigned integer. A similar case occurs with mutations that replace < with ≠ in the conditions of for loops. The test suite cannot distinguish between these cases and reports them as uncaught. • Some of the coverage gaps identified by mutation testing are already covered by the fork and invariant tests that were excluded from the campaign. The decision to exclude these tests was due to the campaign's excessive runtime when using the full test suite. • The surviving mutants in the constructor show that all tests deploy vaults with the same parameters. There are no tests that cover deployments that fail due to incorrect parameters. • Surviving mutants that replace if conditions with constant true or false usually indicate a lack of adversarial inputs in the test suite: only one branch of the conditional is tested.

HashEye 33 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

• The number of surviving mutants in _computeTwapHopQuote shows that the unit tests for the floor calculations are not enough to cover all paths. It is recommended to develop new tests to cover different tick values and different token orders. Mutation Testing Output Figure E.4 shows the full output from the mutation testing campaign before fixes were applied, and figure E.5 shows the output after fixes were applied. In green are lines that correspond to false positives. Target: src/BuybackVault.sol Uncaught | [COS 765] Line 86: 'if (_aiToken == address(0)) revert ZeroAddress();' → 'if (_aiToken ⩽ address(0)) revert ZeroAddress();' Uncaught | [COS 770] Line 87: 'if (_treasury == address(0)) revert ZeroAddress();' → 'if (_treasury ⩽ address(0)) revert ZeroAddress();' Uncaught | [COS 775] Line 88: 'if (_swapRouter == address(0)) revert ZeroAddress();' → 'if (_swapRouter ⩽ address(0)) revert ZeroAddress();' Uncaught | [COS 782] Line 89: 'if (uint256(_burnBps) + uint256(_executorRewardBps) > BPS_DENOMINATOR) revert BpsOverflow();' → 'if (uint256(_burnBps) + uint256(_executorRewardBps) ⩾ BPS_DENOMINATOR) revert BpsOverflow();' Uncaught | [COS 784] Line 90: 'if (_twapWindow < 1800) revert TwapWindowTooShort();' → 'if (_twapWindow ≠ 1800) revert TwapWindowTooShort();' Uncaught | [COS 792] Line 91: 'if (_maxSlippageBps > 500) revert SlippageTooHigh();' → 'if (_maxSlippageBps ⩾ 500) revert SlippageTooHigh();' Uncaught | [COS 795] Line 92: 'if (_owner == address(0)) revert ZeroAddress();' → 'if (_owner ⩽ address(0)) revert ZeroAddress();' Uncaught | [CR 473] Line 95: '__Ownable2Step_init();' → '/* __Ownable2Step_init(); */' Uncaught | [CR 474] Line 96: '__Pausable_init();' → '/* __Pausable_init(); */' constructor Uncaught | [CR 483] Line 106: 'epochStart = block.timestamp.toUint32();' → '/* epochStart = block.timestamp.toUint32(); */' Uncaught | [COS 800] Line 138: 'if (amountIn == 0) revert ZeroAmount();' → 'if (amountIn ⩽ 0) revert ZeroAmount();' Uncaught | [COS 805] Line 139: 'if (amountOutMin == 0) revert ZeroAmount();' → 'if (amountOutMin ⩽ 0) revert ZeroAmount();' Uncaught | [COS 821] Line 141: 'if (path.length < 43 || (path.length - 20) % 23 ≠ 0) revert InvalidPath();' → 'if (path.length < 43 || (path.length - 20) % 23 > 0) revert InvalidPath();' Uncaught | [COS 825] Line 150: 'if (tokenIn == address(0)) {' → 'if (tokenIn ⩽ address(0)) {' Uncaught | [COS 830] Line 152: 'if (_weth == address(0)) revert WethNotConfigured();' → 'if (_weth ⩽ address(0)) revert WethNotConfigured();' Uncaught | [COS 836] Line 165: 'if (pathFirstToken ≠ effectiveTokenIn) revert TokenInMismatch();' → 'if (pathFirstToken > effectiveTokenIn) revert TokenInMismatch();' Uncaught | [COS 8319] Line 166: 'if (pathLastToken ≠ aiToken) revert InvalidPathOutput();' → 'if (pathLastToken < aiToken) revert InvalidPathOutput();' Uncaught | [COS 845] Line 177: 'if (pools.length == 0) revert PoolsLengthMismatch();' → 'if (pools.length ⩽ 0) revert PoolsLengthMismatch();' Uncaught | [COS 855] Line 189: 'if (tokenIn == address(0)) {' → 'if (tokenIn ⩽ address(0)) {'

HashEye 34 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Uncaught | [CR 514] Line 201: 'IERC20(effectiveTokenIn).forceApprove(_swapRouter, 0);' → '/* IERC20(effectiveTokenIn).forceApprove(_swapRouter, 0); */' Uncaught | [IT 655] Line 218: 'if (executorReward > 0) IERC20(_aiToken).safeTransfer(msg.sender, executorReward);' → 'if (true) IERC20(_aiToken).safeTransfer(msg.sender, executorReward);' Uncaught | [IT 656] Line 219: 'if (burnAmount > 0) IBurnable(_aiToken).burn(burnAmount);' → 'if (true) IBurnable(_aiToken).burn(burnAmount);' Uncaught | [IT 657] Line 220: 'if (treasuryAmount > 0) IERC20(_aiToken).safeTransfer(treasury, treasuryAmount);' → 'if (true)

IERC20(_aiToken).safeTransfer(treasury, treasuryAmount);' Uncaught | [COS 881] Line 224: 'if (path.length < 43 || (path.length - 20) % 23 ≠ 0) revert InvalidPath();' → 'if (path.length < 43 || (path.length - 20) % 23 > 0) revert InvalidPath();' Uncaught | [CR 525] Line 226: 'if (pools.length == 0) revert PoolsLengthMismatch();' → '/* if (pools.length == 0) revert PoolsLengthMismatch(); */' Uncaught | [IF 611] Line 226: 'if (pools.length == 0) revert PoolsLengthMismatch();' → 'if (false) revert PoolsLengthMismatch();' Uncaught | [COS 889] Line 229: 'if (pools.length ≠ numHops) revert PoolsLengthMismatch();' → 'if (pools.length < numHops) revert PoolsLengthMismatch();' Uncaught | [COS 894] Line 235: 'if (pathLastToken ≠ aiToken) revert InvalidPathOutput();' → 'if (pathLastToken < aiToken) revert InvalidPathOutput();' Uncaught | [COS 899] Line 240: 'for (uint256 i = 0; i < numHops; i++) {' → 'for (uint256 i = 0; i ≠ numHops; i++) {' Uncaught | [COS 905] Line 251: 'if (hopPool == address(0)) revert ZeroAddress();' → 'if (hopPool ⩽ address(0)) revert ZeroAddress();' Uncaught | [COS 910] Line 253: 'hopTokenIn < hopTokenOut ? (hopTokenIn, hopTokenOut) : (hopTokenOut, hopTokenIn);' → 'hopTokenIn ⩽ hopTokenOut ? (hopTokenIn, hopTokenOut) : (hopTokenOut, hopTokenIn);' Uncaught | [IF 615] Line 254: 'if (IUniswapV3Pool(hopPool).token0() ≠ sortedA) revert PoolMismatch();' → 'if (false) revert PoolMismatch();' Uncaught | [COS 921] Line 255: 'if (IUniswapV3Pool(hopPool).token1() ≠ sortedB) revert PoolMismatch();' → 'if (IUniswapV3Pool(hopPool).token1() > sortedB) revert PoolMismatch();' Uncaught | [COS 924] Line 256: 'if (IUniswapV3Pool(hopPool).fee() ≠ hopFee) revert PoolMismatch();' → 'if (IUniswapV3Pool(hopPool).fee() < hopFee) revert PoolMismatch();' Uncaught | [COS 930] Line 281: 'if (_aiToken == address(0)) revert ZeroAddress();' → 'if (_aiToken ⩽ address(0)) revert ZeroAddress();' Uncaught | [COS 935] Line 287: 'if (_treasury == address(0)) revert ZeroAddress();' → 'if (_treasury ⩽ address(0)) revert ZeroAddress();' Uncaught | [COS 940] Line 293: 'if (_router == address(0)) revert ZeroAddress();' → 'if (_router ⩽ address(0)) revert ZeroAddress();' Uncaught | [CR 554] Line 324: 'epochStart = block.timestamp.toUint32();' → '/* epochStart = block.timestamp.toUint32(); */' setEpochConfig Uncaught | [CR 555] Line 325: 'epochIndex++;' → '/* epochIndex++; */' Uncaught | [COS 971] Line 335: 'if (_weth ≠ address(0) && _weth.code.length == 0) revert NotAContract();' → 'if (_weth > address(0) && _weth.code.length == 0) revert NotAContract();'

HashEye 35 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Uncaught | [COS 965] Line 335: 'if (_weth ≠ address(0) && _weth.code.length == 0) revert NotAContract();' → 'if (_weth ≠ address(0) && _weth.code.length ⩽ 0) revert NotAContract();' Uncaught | [COS 975] Line 349: 'if (to == address(0)) revert ZeroAddress();' → 'if (to ⩽ address(0)) revert ZeroAddress();' Uncaught | [COS 980] Line 350: 'if (token == address(0)) {' → 'if (token ⩽ address(0)) {' Uncaught | [COS 985] Line 360: 'if (epochDuration == 0) return;' → 'if (epochDuration ⩽ 0) return;' Uncaught | [AOS 733] Line 362: 'if (block.timestamp ⩾ uint256(epochStart) + uint256(epochDuration)) {' → 'if (block.timestamp ⩾ uint256(epochStart) * uint256(epochDuration)) {' Uncaught | [COS 995] Line 368: 'if (limit == 0) return;' → 'if (limit ⩽ 0) return;' Uncaught | [COS 1002] Line 370: 'uint256 currentVolume = tokenEpochIndex[tokenIn] == epochIndex ? tokenEpochVolume[tokenIn] : 0;' → 'uint256 currentVolume = tokenEpochIndex[tokenIn] ⩾ epochIndex ? tokenEpochVolume[tokenIn] : 0;' Uncaught | [CR 571] Line 374: 'tokenEpochIndex[tokenIn] = epochIndex;' → '/* tokenEpochIndex[tokenIn] = epochIndex; */' Uncaught | [COS 1009] Line 386: 'for (uint256 i = 0; i < numHops; i++) {' → 'for (uint256 i = 0; i ≠ numHops; i++) {' Uncaught | [AOS 738] Line 388: 'uint256 hopOffset = i * 23;' → 'uint256 hopOffset = i + 23;' Uncaught | [AOS 740] Line 388: 'uint256 hopOffset = i * 23;' → 'uint256 hopOffset = i / 23;' Uncaught | [CR 579] Line 405: 'secondsAgos[0] = window;' → '/* secondsAgos[0] = window; */' Uncaught | [CR 580] Line 406: 'secondsAgos[1] = 0;' → '/* secondsAgos[1] = 0; */' Uncaught | [AOS 750] Line 409: 'int56 tickDelta = tickCumulatives[1] - tickCumulatives[0];' → 'int56 tickDelta = tickCumulatives[1] + tickCumulatives[0];' Uncaught | [CR 584] Line 411: 'if (tickDelta < 0 && (tickDelta % int56(uint56(window)) ≠ 0)) meanTick--;' → '/* if (tickDelta < 0 && (tickDelta % int56(uint56(window)) ≠ 0)) meanTick--; */' Uncaught | [IF 633] Line 411: 'if (tickDelta < 0 && (tickDelta % int56(uint56(window)) ≠ 0)) meanTick--;' → 'if (false) meanTick--;' Uncaught | [IF 634] Line 415: 'if (sqrtRatioX96 ⩽ type(uint128).max) {' → 'if (false) {' Uncaught | [COS 1030] Line 417: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn ⩽ tokenOut' Uncaught | [COS 1028] Line 417: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn == tokenOut' Uncaught | [COS 1034] Line 422: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn ≠ tokenOut' Uncaught | [COS 1035] Line 422: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn ⩽ tokenOut' Uncaught | [COS 1033] Line 422: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn == tokenOut' Uncaught | [COS 1036] Line 422: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn > tokenOut' Uncaught | [COS 1037] Line 422: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn ⩾ tokenOut'

HashEye 36 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Uncaught | [SOS 1041] Line 423: '? Math.mulDiv(ratioX128, amountIn, 1 << 128)' → '? Math.mulDiv(ratioX128, amountIn, 1 >> 128)' Uncaught | [SOS 1042] Line 424: ': Math.mulDiv(1 << 128, amountIn, ratioX128);' → ': Math.mulDiv(1 >> 128, amountIn, ratioX128);' High severity caught: 72.4% (89 / 123) Medium severity caught: 91.8% (201 / 219) Low severity caught: 85.0% (271 / 319) Total caught: 84.9% (561 / 661)

Figure E.4: Complete output from the mutation testing campaign before the fixes were applied Target: src/BuybackVault.sol Uncaught | [COS 413] Line 89: 'if (_aiToken == address(0)) revert ZeroAddress();' → 'if (_aiToken ⩽ address(0)) revert ZeroAddress();' Uncaught | [COS 418] Line 90: 'if (_treasury == address(0)) revert ZeroAddress();' → 'if (_treasury ⩽ address(0)) revert ZeroAddress();' Uncaught | [COS 423] Line 91: 'if (_swapRouter == address(0)) revert ZeroAddress();' → 'if (_swapRouter ⩽ address(0)) revert ZeroAddress();' Uncaught | [COS 430] Line 92: 'if (uint256(_burnBps) + uint256(_executorRewardBps) > BPS_DENOMINATOR) revert BpsOverflow();' → 'if (uint256(_burnBps) + uint256(_executorRewardBps) ⩾ BPS_DENOMINATOR) revert BpsOverflow();' Uncaught | [COS 432] Line 93: 'if (_twapWindow < 1800) revert TwapWindowTooShort();' → 'if (_twapWindow ≠ 1800) revert TwapWindowTooShort();' Uncaught | [COS 440] Line 94: 'if (_maxSlippageBps > 500) revert SlippageTooHigh();' → 'if (_maxSlippageBps ⩾ 500) revert SlippageTooHigh();' Uncaught | [COS 443] Line 95: 'if (_owner == address(0)) revert ZeroAddress();' → 'if (_owner ⩽ address(0)) revert ZeroAddress();' Uncaught | [CR 132] Line 98: '__Ownable2Step_init();' → '/* __Ownable2Step_init(); */' Uncaught | [CR 133] Line 99: '__Pausable_init();' → '/* __Pausable_init(); */' Uncaught | [CR 142] Line 109: 'epochStart = block.timestamp.toUint32();' → '/* epochStart = block.timestamp.toUint32(); */' Uncaught | [COS 448] Line 141: 'if (amountIn == 0) revert ZeroAmount();' → 'if (amountIn ⩽ 0) revert ZeroAmount();' Uncaught | [COS 453] Line 142: 'if (amountOutMin == 0) revert ZeroAmount();' → 'if (amountOutMin ⩽ 0) revert ZeroAmount();' Uncaught | [COS 469] Line 144: 'if (path.length < 43 || (path.length - 20) % 23 ≠ 0) revert InvalidPath();' → 'if (path.length < 43 || (path.length - 20) % 23 > 0) revert InvalidPath();' Uncaught | [COS 473] Line 153: 'if (tokenIn == address(0)) {' → 'if (tokenIn ⩽ address(0)) {' Uncaught | [COS 478] Line 156: 'if (_weth == address(0)) revert WethNotConfigured();' → 'if (_weth ⩽ address(0)) revert WethNotConfigured();' Uncaught | [COS 484] Line 169: 'if (pathFirstToken ≠ effectiveTokenIn) revert TokenInMismatch();' → 'if (pathFirstToken > effectiveTokenIn) revert TokenInMismatch();' Uncaught | [COS 487] Line 170: 'if (pathLastToken ≠ aiToken) revert InvalidPathOutput();' → 'if (pathLastToken < aiToken) revert InvalidPathOutput();' Uncaught | [COS 493] Line 181: 'if (pools.length == 0) revert PoolsLengthMismatch();' → 'if (pools.length ⩽ 0) revert PoolsLengthMismatch();'

HashEye 37 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Uncaught | [COS 503] Line 193: 'if (tokenIn == address(0)) {' → 'if (tokenIn ⩽ address(0)) {' Uncaught | [CR 173] Line 205: 'IERC20(effectiveTokenIn).forceApprove(_swapRouter, 0);' → '/* IERC20(effectiveTokenIn).forceApprove(_swapRouter, 0); */' Uncaught | [IT 309] Line 222: 'if (executorReward > 0) IERC20(_aiToken).safeTransfer(msg.sender, executorReward);' → 'if (true) IERC20(_aiToken).safeTransfer(msg.sender, executorReward);' Uncaught | [IT 310] Line 223: 'if (burnAmount > 0) IBurnable(_aiToken).burn(burnAmount);' → 'if (true) IBurnable(_aiToken).burn(burnAmount);' Uncaught | [IT 311] Line 224: 'if (treasuryAmount > 0) IERC20(_aiToken).safeTransfer(treasury, treasuryAmount);' → 'if (true) IERC20(_aiToken).safeTransfer(treasury, treasuryAmount);' Uncaught | [COS 529] Line 228: 'if (path.length < 43 || (path.length - 20) % 23 ≠ 0) revert InvalidPath();' → 'if (path.length < 43 || (path.length - 20) % 23 > 0) revert InvalidPath();' Uncaught | [COS 532] Line 236: 'if (pathLastToken ≠ aiToken) revert InvalidPathOutput();' → 'if (pathLastToken < aiToken) revert InvalidPathOutput();' Uncaught | [COS 537] Line 241: 'for (uint256 i = 0; i < numHops; i++) {' → 'for (uint256 i = 0; i ≠ numHops; i++) {' Uncaught | [COS 543] Line 252: 'if (hopPool == address(0)) revert PoolNotFound();' → 'if (hopPool ⩽ address(0)) revert PoolNotFound();' Uncaught | [COS 548] Line 281: 'if (_treasury == address(0)) revert ZeroAddress();' → 'if (_treasury ⩽ address(0)) revert ZeroAddress();' Uncaught | [COS 553] Line 287: 'if (_router == address(0)) revert ZeroAddress();' → 'if (_router ⩽ address(0)) revert ZeroAddress();' Uncaught | [CR 211] Line 318: 'epochStart = block.timestamp.toUint32();' → '/* epochStart = block.timestamp.toUint32(); */' Uncaught | [CR 212] Line 319: 'epochIndex++;' → '/* epochIndex++; */' Uncaught | [COS 584] Line 329: 'if (_weth ≠ address(0) && _weth.code.length == 0) revert NotAContract();' → 'if (_weth > address(0) && _weth.code.length == 0) revert NotAContract();'

Uncaught | [COS 578] Line 329: 'if (_weth ≠ address(0) && _weth.code.length == 0) revert NotAContract();' → 'if (_weth ≠ address(0) && _weth.code.length ⩽ 0) revert NotAContract();' Uncaught | [COS 588] Line 348: 'if (to == address(0)) revert ZeroAddress();' → 'if (to ⩽ address(0)) revert ZeroAddress();' Uncaught | [COS 593] Line 349: 'if (token == address(0)) {' → 'if (token ⩽ address(0)) {' Uncaught | [COS 598] Line 359: 'if (epochDuration == 0) return;' → 'if (epochDuration ⩽ 0) return;' Uncaught | [COS 605] Line 361: 'if (block.timestamp ⩾ uint256(epochStart) + uint256(epochDuration)) {' → 'if (block.timestamp > uint256(epochStart) + uint256(epochDuration)) {' Uncaught | [AOS 381] Line 361: 'if (block.timestamp ⩾ uint256(epochStart) + uint256(epochDuration)) {' → 'if (block.timestamp ⩾ uint256(epochStart) * uint256(epochDuration)) {' Uncaught | [COS 608] Line 367: 'if (limit == 0) return;' → 'if (limit ⩽ 0) return;' Uncaught | [COS 615] Line 369: 'uint256 currentVolume = tokenEpochIndex[tokenIn] == epochIndex ? tokenEpochVolume[tokenIn] : 0;' → 'uint256 currentVolume = tokenEpochIndex[tokenIn] ⩾ epochIndex ? tokenEpochVolume[tokenIn] : 0;'

HashEye 38 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

Uncaught | [CR 229] Line 373: 'tokenEpochIndex[tokenIn] = epochIndex;' → '/* tokenEpochIndex[tokenIn] = epochIndex; */' Uncaught | [COS 622] Line 385: 'for (uint256 i = 0; i < numHops; i++) {' → 'for (uint256 i = 0; i ≠ numHops; i++) {' Uncaught | [AOS 386] Line 387: 'uint256 hopOffset = i * 23;' → 'uint256 hopOffset = i + 23;' Uncaught | [AOS 388] Line 387: 'uint256 hopOffset = i * 23;' → 'uint256 hopOffset = i / 23;' Uncaught | [CR 237] Line 404: 'secondsAgos[0] = window;' → '/* secondsAgos[0] = window; */' Uncaught | [CR 238] Line 405: 'secondsAgos[1] = 0;' → '/* secondsAgos[1] = 0; */' Uncaught | [AOS 398] Line 408: 'int56 tickDelta = tickCumulatives[1] - tickCumulatives[0];' → 'int56 tickDelta = tickCumulatives[1] + tickCumulatives[0];' Uncaught | [CR 242] Line 410: 'if (tickDelta < 0 && (tickDelta % int56(uint56(window)) ≠ 0)) meanTick--;' → '/* if (tickDelta < 0 && (tickDelta % int56(uint56(window)) ≠ 0)) meanTick--; */' Uncaught | [IF 286] Line 410: 'if (tickDelta < 0 && (tickDelta % int56(uint56(window)) ≠ 0)) meanTick--;' → 'if (false) meanTick--;' Uncaught | [IF 287] Line 414: 'if (sqrtRatioX96 ⩽ type(uint128).max) {' → 'if (false) {' Uncaught | [COS 643] Line 416: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn ⩽ tokenOut' Uncaught | [COS 641] Line 416: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn == tokenOut' Uncaught | [COS 647] Line 421: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn ≠ tokenOut' Uncaught | [COS 648] Line 421: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn ⩽ tokenOut' Uncaught | [COS 646] Line 421: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn == tokenOut' Uncaught | [COS 649] Line 421: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn > tokenOut' Uncaught | [COS 650] Line 421: 'amountOut = tokenIn < tokenOut' → 'amountOut = tokenIn ⩾ tokenOut' Uncaught | [SOS 654] Line 422: '? Math.mulDiv(ratioX128, amountIn, 1 << 128)' → '? Math.mulDiv(ratioX128, amountIn, 1 >> 128)' Uncaught | [SOS 655] Line 423: ': Math.mulDiv(1 << 128, amountIn, ratioX128);' → ': Math.mulDiv(1 >> 128, amountIn, ratioX128);' High severity caught: 100.0% (122 / 122) Medium severity caught: 92.8% (193 / 208) Low severity caught: 85.0% (250 / 294) Total caught: 90.5% (565 / 624) Figure E.5: Complete output from the mutation testing campaign after the fixes were applied Mutation Testing Conclusions

We recommend that the Gensyn team thoroughly review and evaluate the mutation test results and fix the test suite for all uncaught mutants, and then rerun a mutation testing campaign to ensure that the added or modified tests provide adequate coverage. Alternatively, run a mutation testing campaign with the full test suite and compare the results to validate that fork, fuzz, or invariant tests cover the gaps found in unit tests.

HashEye 39 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

F. Incident Response Recommendations This section provides recommendations on formulating an incident response plan. Also, our secure-contracts website has more information about incident response recommendations and useful resources. • Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.). • Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug. ○ Consider documenting a plan of action for handling failed remediations. • Clearly describe the intended contract deployment process. • Outline the circumstances under which Gensyn will compensate users affected by an issue (if any). ○ Issues that warrant compensation could include an individual or

aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw. ● Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on. ○ Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific system components. ● Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers) and how it will onboard them. ○ Effective remediation of certain issues may require collaboration with external parties.

HashEye 40 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

● Define contract behavior that would be considered abnormal by off-chain monitoring solutions. It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop “muscle memory.” Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

HashEye 41 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

About HashEye Founded in 2012 and headquartered in New York, HashEye provides technical security assessment and advisory services to some of the world’s most targeted organizations. We combine high- end security research with a real -world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we’ve helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel. We maintain an exhaustive list of publications at https://github.com/hasheye-io/publications, with links to papers, presentations, public audit reports, and podcast appearances. In recent years, HashEye consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O’Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon. We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom. To keep up with our latest news and announcements, please follow hasheye on X or LinkedIn and explore our public repositories at https://github.com/hasheye-io. To engage us directly, visit our “Contact” page at https://www.hasheye.io/contact or email us at info@hasheye.io. HashEye, Inc. 228 Park Ave S #80688 New York, NY 10003 https://www.hasheye.io info@hasheye.io

HashEye 42 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment

 Test Coverage Disclaimer

HashEye performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan. Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase. HashEye uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review

work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.

HashEye 43 Gensyn Buyback-and-Burn Vault

PUBLIC Security Assessment