

## Nested Finance

Security assessment by HashEye · prepared for Nested Finance

HASHEYE AUDITED

PROJECT	Nested Finance
CLIENT	Nested Finance
CATEGORY	Blockchain
PUBLISHED	February 1, 2023
REPORT ID	research-nested-finance-2023-02-01-1wp9a6

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 [hashey.io/audits/research-nested-finance-2023-02-01-1wp9a6](https://hashey.io/audits/research-nested-finance-2023-02-01-1wp9a6).

Mass SecurityAssessment April28,2023 Preparedfor: RudyKadoch Mass rk@mass.money OlivierGuimbal Mass og@mass.money Preparedby:JosselinFeist,GustavoGrieco,KurtWillis,TarunBansal,andRichie Humphrey

About hashey Founded in 2012 and headquartered in New York, hashey 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/hashey-io/publications>, with links to papers, presentations, public audit reports, and podcast appearances. In recent years, hashey 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 projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom. hashey also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow hashey on Twitter and explore our public repositories at <https://github.com/hashey-io>. To engage us directly, visit our "Contact" page at <https://www.hashey.io/contact>, or email us at [info@hashey.io](mailto:info@hashey.io). hashey, Inc. 228 Park Ave S #80688 New York, NY 10003 <https://www.hashey.io> [info@hashey.io](mailto:info@hashey.io) hashey 1 Nested Finance Security Assessment PUBLIC

Notices and Remarks Copyright and Distribution ©2023 by hashey, Inc. All rights reserved. hashey hereby asserts its right to be identified as the creator of this report in the United Kingdom. This report is considered by hashey to be public information; it is licensed to Nested Finance under the terms of the project statement of work and has been made public at Nested Finance's request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of hashey.

This sole canonical source for hashey publications is the hashey Publications page. Reports accessed through any source other than that page may have been modified and should not be considered authentic. Test Coverage Disclaimer

All activities undertaken by hashey in association with this project were performed in accordance with a statement of work and agreed upon project plan. Security assessment projects are time-boxed and often reliant on information that may be 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 code base.

hashey uses automated testing techniques to rapidly test the controls and security properties of software. 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. Their use is also limited by the time and resource constraints of a project. hashey 2 Nested Finance Security Assessment PUBLIC

Table of Contents About hashey 1 Notices and Remarks 2 Table of Contents 3 Executive Summary 5 Project Summary 7 Project Goals 8 Project Targets 9 Project Coverage 10 Automated Testing 12 Codebase Maturity Evaluation 14 Summary of Findings: HyVM 17 Detailed Findings: HyVM 18  
1. Unprotected call code allows anyone to destroy the HyVM master contact 18  
2. Lack of contract existence check on low-level calls 20  
7. Risk of overflow in FIX\_MEMOFFSET that could allow attacker to overwrite protected memory 22  
8. Users can directly call the HyVM implementation 24 9. State-changing operations permitted in HyVM 25  
10. Lack of completed documentation on the known HyVM/EVM divergences 26 11. Outdated HyVM used by nested-core-tetris 27 12. Risks associated with using the Huff programming language 29 Summary of Findings: Tetris Core 31  
Detailed Findings: Tetris Core 32 13. Timelock Controller Emergency could be used to allow anyone to execute proposals 32 14. Risks associated with the use and design of the Huff proxy 35  
15. Use of tx.origin for access controls 38 16. Ability to create wallets before WalletFactory is initialized 40  
Summary of Findings: Token 42 Detailed Findings: Token 43 17. Risk of portfolio voting power theft 43  
18. acceptDeal adds unreleased token to new vesting schedules 45

19. Vesting contract owner is not updated when Vesting Factory owner changes 47  
20. Unbounded loops could cause denial of service for third-party integrations 49 hashey  
3 Nested Finance Security Assessment PUBLIC

Summary of Findings: DCA51 Detailed Findings: DCA52 21. DCA ownership transfers can be abused to steal tokens 52  
22. Risk of unlimited slippage in Nested DCA swaps 54  
23. Insufficient DCA existence check could result in an inconsistent state 56  
24. Users prevented from using special ETH address to pay Gelato fees 58 25. DCA tasks can be stopped by anyone 60  
26. DCA task IDs can be manually set 62 27. restart DCA lacks data validation 64  
28. Reentrancy vulnerabilities in Nested DCA could allow Gelato operator to drain left over tokens 66  
29. Unclear usage of DCA.tokensOut Allocation 68 A. Vulnerability Categories 70 B. Code Maturity Categories 72  
C. Recommendations for Nested DCA's Data Schema 74 D. Code Quality Findings 76 E. Fix Review Results 78  
Detailed Fix Review Results 80 hashey 4 Nested Finance Security Assessment PUBLIC

#### Executive Summary Engagement Overview

Nested Finance engaged hashey to review the security of its HyVM, Tetris, Nested DCA, and Nested token codebases. From January 30 to February 10, 2023, a team of five consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Our testing efforts focused on identifying issues that would allow attackers to steal a wallet's funds, break the HyVM's execution, abuse the dollar cost averaging (DCA) protocol, or steal vested tokens. With access to the source code and documentation, we performed static and dynamic testing of the target system, using automated and manual processes. After discovering several findings related to the DCA contracts in the first part of the audit, we worked with Nested Finance to prioritize reviewing the HyVM, Tetris, and token codebases for the remainder of the audit. Observations and Impact In developing the target codebases, Nested Finance seem to have prioritized modularity, support for multiple use cases, and optimizations; this focus has increased the codebases' complexity and related risks across most of the components. During our review, we identified many issues related to the following, indicating that the codebases would benefit from improved testing and data structure design:

- Improper use of data structures and validation
- Lack of clarity and expectations with regard to access controls
- Undefined expectation over the contracts' state transitions (e.g., lifecycle of a DCA)

During the audit, we discovered several significant findings across most of the components, including the following:

- The lack of access controls on the call code opcode, allowing anyone to destroy the HyVM contract (TOB-NESTED-1)
- The risk of memory corruption in the HyVM due to a possible pointer overflow (TOB-NESTED-7)
- The risk of theft of a portfolio's voting power due to incorrect bookkeeping (TOB-NESTED-17)
- The risk of theft of tokens in the DCA contract due to improper access controls (TOB-NESTED-21) hashey

5 Nested Finance Security Assessment PUBLIC

Through our review, we found that the testing and specification of the HyVM codebase does not meet the maturity requirements for a component written entirely in raw opcodes. Moreover, the DCA codebase appears to require further development, specifically to address data structure issues. Recommendations hashey recommends that Nested Finance address the findings detailed in this report and take the following additional steps:

- Change the design paradigm to strive for simplicity over modularity and more general support of multiple use cases.
- Create Solidity reference implementations for any component that is optimized through Huff.

Evaluate the HyVM features that are needed for creating and managing Nested wallets, and focus the development of the HyVM based on those needs.

- Define the expected state transitions of the DCA and ensure that the implementation follows the specification.

The following tables provide the number of findings by severity and category. EXPOSURE ANALYSIS

Severity	Count
High	6
Medium	3
Low	12

Informational	8	
CATEGORY BREAKDOWN	Category	Count
Access Controls	2	
Authentication	1	
Configuration	1	
Data Validation	18	
Patching	1	
Undefined Behavior	6	

hashey

6 Nested Finance Security Assessment PUBLIC

Project Summary Contact Information The following managers were associated with this project:  
Dan Guido, Account Manager Mary O'Brien, Project Manager dan@hashey.io mary.obrien@hashey.io  
The following engineers were associated with this project:  
Josselin Feist, Consultant Gustavo Grieco, Consultant josselin@hashey.io gustavo.grieco@hashey.io  
Kurt Willis, Consultant Tarun Bansal, Consultant kurt.willis@hashey.io tarun.bansal@hashey.io  
Richie Humphrey, Consultant richie.humphrey@hashey.io Project Timeline  
The significant events and milestones of the project are listed below.

Date	Event
January 26, 2023	Pre-project kickoff call
February 6, 2023	Status update meeting #1
February 13, 2023	Delivery of report draft
February 13, 2023	Report readout meeting
March 20, 2023	Delivery of final report
April 28, 2023	Delivery of final report with fix review (appendix E) hashey

7 Nested Finance Security Assessment PUBLIC

Project Goals The engagement is scoped to provide a security assessment of Nested Finance's sHyVM, Tetris, NestedDCA, and NestedToken codebases. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can an attacker steal or freeze funds?
- Are there appropriate access control measures in place for users and owners?
- Does the system's behavior match the specification?
- Is it possible to perform system operations without paying the required amount?
- Are there any undocumented discrepancies between the use of the HyVM and EVM?
- Is HyVM's memory management safe?
- Are there any risks of using the Huff proxy?
- Can users withdraw more from Vesting contracts than expected?
- Can a Vesting contract's owner prevent the withdrawal of tokens?
- Can users lose money from using the DCA contracts?
- Can an attacker prevent the execution of a DCA strategy?
- Are all of the common Solidity flaws avoided?

hashey 8NestedFinanceSecurityAssessment PUBLIC

Project Targets The engagement involved a review and testing of the targets listed below.

HyVM  
Repository <https://github.com/aguimbal/HyVM> Version 4e760d4535cf0e55d4453583c9b1484e5f3d3f9c  
Type Huff Platform Ethereum Tetris  
Repository <https://github.com/NestedFi/nested-core-tetris> Version af206e2d1e6070574bfff5c2639569812a260a7f5  
Types Solidity, Huff Platform Ethereum NestedDCA  
Repository <https://github.com/NestedFi/nested-dca> Version cdaaedb4612f5576d644b8032350e4fce40f1ba1  
Type Solidity Platform Ethereum NestedToken  
Repository <https://github.com/NestedFi/nested-token>  
Version 31e0d470277c6e20f5fe1e537635d98fa1328980  
Type Solidity Platform Ethereum hashey 9NestedFinanceSecurityAssessment PUBLIC

Project Coverage This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- **HyVM:** This component is an Ethereum Virtual Machine (EVM) Hypervisor written in Huff that allows users to execute arbitrary EVM bytecode. This component replaces the use of specific scripts to interact with external protocols. We manually reviewed the HyVM to look for potential divergences from the EVM. We reviewed the memory model of the HyVM, with a focus on finding ways to break the protected memory. We looked at how the jump table is constructed and used. Finally, we performed differential tests to detect further divergences from the EVM's execution.

- **Tetris:** This component is an account abstraction protocol that enables the creation of smart contract wallets, called "Nested wallets" on the Nested platform. Tetris allows users to interact with any protocol and store their funds in their wallets all in a single transaction. We manually reviewed the creation of smart contract wallets and the implementation of their access controls. Additionally, we looked at their architectural risks and looked for any potentially problematic interactions with other systems, either through the HyVM or direct calls. We reviewed Tetris's proxy contract, written in Huff, and looked for incorrect stack/memory manipulations and opportunities to perturb the use of the resolver.

- **NestedToken:** This component contains the contracts related to the Nested platform's native ERC-20 token, NST, implemented using OpenZeppelin contracts.

NST tokens can be vested and delegated. We manually reviewed the token's capabilities and features to ensure that users cannot disrupt transferring, voting, or vesting processes. We reviewed the vesting formula and looked for ways to release more tokens than expected, such as by abusing time limitations or deals. We also searched for opportunities for a malicious Vesting contract owner to prevent the withdrawal of tokens. Finally, we used automated testing to identify and check properties related to the vesting procedure.

- **NestedDCA:** This component is a DCA protocol that allows users to invest a fixed dollar amount in one or more tokens, regardless of their prices. Users can program automated recurring swaps, called tasks, which will then be executed by the Gelato Network. Users only need to ensure that they have sufficient funds to cover the investment and the automation fee. We manually reviewed the internal bookkeeping of tasks and its access controls to ensure its correct creation, execution, and removal of swaps. Additionally, we partially reviewed the DCA component's integration with third-party automation systems like Gelato. Because we found a large number of issues related to the DCA during the first week of the review, Nested Finance decided to reduce the priority of this component. The DCA

hashey 10NestedFinanceSecurityAssessment PUBLIC

contracts were not covered during the second week of the review. Therefore, additional issues are likely to be present.

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:

- Certain components such as the NestedDCA and the NST's vesting procedure had

incomplete or unclear specifications/documentation, which resulted in lower coverage of complex interactions during the manual review process. • Four issues were found in the contract-verifier component of the HyVM but are not present in this report, as Nested Finance classified this component as out of scope. Additional issues are likely present in the verifier. The following are the four contract-verifier issues: ◦ TOB-NESTED-3: CHECK\_DELEGATECALL and CHECK\_CALLCODE push incorrect opcodes ◦ TOB-NESTED-4: Missing check of verifyCall 's return data size ◦ TOB-NESTED-5: CHECK\_SSTORE incorrectly checks values instead of keys ◦ TOB-NESTED-6: CHECK\_SSTORE does not check for ERC\_2771\_ACTIVATED\_SLOT • Third-party components such as OpenZeppelin and Gelato contracts were not reviewed. • The Huff programming language compiler was out of scope, so its security and correctness were not reviewed. • Previously known issues (e.g., the incorrect validation of JUMPDEST in the HyVM contract) are not present in this report. hashey 11 Nested Finance Security Assessment PUBLIC

Automated Testing hashey 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 Slither A static analysis framework that can statically verify algebraic relationships between Solidity variables

Echidna A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation Test Results The results of this focused testing are detailed below. Vesting Contract Property Tool Result If the preconditions are met, the setLeaver function never reverts. Echidna Passed If the preconditions are met, the acceptDeals function never reverts. Echidna Passed The duration of the last schedule is always strictly positive. Echidna Passed The releasable function never reverts. Echidna Passed hashey 12 Nested Finance Security Assessment PUBLIC

HyVM and EVM Differential Testing Property Tool Result There are no discrepancies between the EVM's and HyVM's executions. Echidna TOB-NESTED -7 TOB-NESTED -10 hashey 13 Nested Finance Security Assessment PUBLIC

Codebase Maturity Evaluation hashey 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 lifecycle 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 Overall, the codebases do not rely on complex arithmetic. Most of the arithmetic-related complexity is located in the vesting and DCA operations. While the vesting process's formula has been documented and some tests are applied to check the arithmetic, the use of a fuzzer—or a similar automated testing method—would increase confidence in the arithmetic operations. For example, a fuzzer could stress test the tokens released over time and ensure that their values match their expectations. Additionally, for operations that lead to a loss of precision, we recommend including additional documentation to indicate that the rounding direction does not benefit users. Finally, the use of unprotected overflow operations in the HyVM contract increases the likelihood of mistakes. Moderate Auditing While there are events for critical components to monitor the system off-chain, there is a lack of documentation on how to do so. Additionally, an incident response plan describing how the protocol's actors must react in case of failure was drafted, but the final version was not available during the review period. Moderate Authentication/ Access Controls Certain components of the codebase have very loose access controls (e.g., DCA tasks, the bridge's voting power, and HyVM direct calls). The roles and names of certain components such as those in the Vesting contract (e.g., team/beneficiary) are confusing and make the controls more difficult to review. Weak Complexity Management Although most of the functions are short and well identified, the codebases suffer from overall complexity Weak hashey 14 Nested Finance Security Assessment PUBLIC

that may be unnecessary. Nested Finance has prioritized genericity and optimization in developing the codebases, which has significantly reduced their simplicity. We identified several components that could be drastically simplified while retaining all the functionalities required by Nested wallets, including the proxies and resolvers (TOB-NESTED-14) and the use of the HyVM itself (TOB-NESTED-8, TOB-NESTED-9). Decentralization While Nested wallets are intended to be decentralized portfolios, the implementation resolver prevents this goal from being reached. The owner(s) of the resolver could change the implementation used for any wallet or factory at any moment. Even if the resolver's owner is a multisig wallet, the multisig owners could collude to make such changes as well. Users currently have no mechanism to opt out of emergency upgrades, and they could lose control over their wallets. Weak Documentation The quality of the documentation is uneven across the codebases. While the proxies' interactions are well documented and have diagrams, the functionalities of the DCA and the Vesting wallet lack clarity. Several features have ambiguous or missing documentation (e.g., the DCA contract's tokensOutAllocation parameter). Due to its low-level complexity and risks, the HyVM

requires a well-defined specification. Weak Ordering Risks  
Although only the DCA protocol was indicated as an area of concern for maximal extractable value (MEV) risks, we found no tests of the protocol's slippage protection to prevent related vulnerabilities. Also, the documentation contains no clear identification of the known MEV risks, and it is unclear whether the risks related to Gelato operators are well understood. Moderate Low-Level Manipulation

The code bases rely heavily on the use of optimizations and assembly; however, their use seems unnecessary in several cases. The documentation and testing practices do not match the maturity required for contracts with this amount of low-level manipulation. For example, we found no tests for some opcodes (including `jump` / `jumpi`) in the HyVM. A reference implementation in Solidity for code that was written in Huff for the sole purpose of Weak hashey 15 Nested Finance Security Assessment PUBLIC

optimizations is missing. Without a reference implementation, advanced testing (e.g., differential fuzzing) cannot be applied, making it more difficult to review the code. Testing and Verification  
Although unit tests are used, they do not cover certain features; as a result, we encountered code that is non-functional or unreachable (e.g., the ability to withdraw voting power in the bridge and to use ETH as a token fee in the DCA protocol). In addition to the lack of unit testing coverage, there are no fuzz tests. These types of tests are particularly useful for ensuring that the EVM can be correctly executed by the HyVM. Weak hashey 16 Nested Finance Security Assessment PUBLIC

#### Summary of Findings: HyVM

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

Detailed Findings: HyVM

ID	Title	Type	Severity
1	Unprotected callcode allows anyone to destroy the HyVM master contract	Data Validation	High
2	Lack of contract existence check on low-level calls	Data Validation	High
7	Risk of overflow in <code>FIX_MEMOFFSET</code> that could allow attacker to overwrite protected memory	Data Validation	High
8	Users can directly call the HyVM implementation	Undefined Behavior	Informational
9	State-changing operations permitted in HyVM	Undefined Behavior	Informational
10	Lack of completed documentation on the known HyVM/EVM divergences	Undefined Behavior	Informational
11	Outdated HyVM used by nested-core-tetris	Undefined Behavior	Informational
12	Risks associated with using the Huff programming language	Patching	Informational

hashey 17 Nested Finance Security Assessment PUBLIC

Detailed Findings: HyVM 1. Unprotected callcode allows anyone to destroy the HyVM master contract  
Severity: High Difficulty: Low Type: Data Validation Finding ID: TOB-NESTED-1 Target: HyVM.huff Description  
The HyVM contract allows unrestricted access to execute the callcode opcode, allowing anyone to destroy the contract.

The HyVM is a virtual machine that simulates EVM execution. The opcodes to be executed are passed through the transaction's call data. Among other EVM opcodes, the HyVM contract allows the execution of callcode : `op_callcode` // takes // [gas, address, value, argsOffset, argsSize, retOffset, retSize] // fix argsOffset `swap3` // [argsOffset, address, value, gas, argsSize, retOffset, retSize] `FIX_MEMOFFSET()` // [hyvmArgsOffset, address, value, gas, argsSize, retOffset, retSize] `swap3` // [gas, address, value, hyvmArgsOffset, argsSize, retOffset, retSize] // fix retOffset `swap5` // [retOffset, address, value, hyvmArgsOffset, argsSize, gas, retSize] `FIX_MEMOFFSET()` // [hyvmRetOffset, address, value, hyvmArgsOffset, argsSize, gas, retSize] `swap5` // [gas, address, value, hyvmArgsOffset, argsSize, hyvmRetOffset, retSize] // check can call `CHECK_CALLCODE()` // [gas, address, value, hyvmArgsOffset, argsSize, hyvmRetOffset, retSize] `callcode` // [success] `CONTINUE()` Figure 1.1: HyVM.huff#L682-L698

This opcode has no restriction. Therefore, anyone can execute callcode to a controlled address and execute `selfdestruct` to destroy the HyVM contract. hashey 18 Nested Finance Security Assessment PUBLIC

Combined with issue TOB-NESTED-2, the impact of this issue is more severe: if HyVM is destroyed and Nested wallets use `delegatecall` to call HyVM, those calls will return success but will not execute any code. Note that the `delegatecall` opcode has the same risk, but this opcode is not executable through a direct call to HyVM. Exploit Scenario  
Bob uses a Nested wallet for his portfolio. Bob has a \$10 million position that is close to being liquidated. Even notice the situation and calls `selfdestruct` through the callcode opcode to destroy the HyVM contract. Bob tries to add liquidity through the execution of a HyVM payload; his transaction succeeds, but no code is executed. As a result, Bob's position is liquidated. Recommendations  
Short term, remove the callcode opcode and document the removal. The callcode opcode is deprecated and generally considered unsafe.

Long term, carefully review the EVM yellow paper and be aware of the semantics of every opcode. Document every opcode used in HyVM and justify their presence. hashey 19 Nested Finance Security Assessment PUBLIC

2. Lack of contract existence check on low-level calls Severity: High Difficulty: High  
Type: Data Validation Finding ID: TOB-NESTED-2 Target: All code bases Description Several of the low-level/assembly calls made in the code bases do not check that their destinations have code; these calls will succeed even if their destinations have no code. For example, all the low-

```

Levelcallsin Proxy.huff lackcontractexistencechecks: #definemacroMAIN()=takes(0)returns(0){
LOAD_IMPLEMENTATION_RESOLVER_CALLDATA()
//[gas,implementationResolverAddress,argsOffset,argsSize,retOffset,retSize] staticcall //[success]
Figure2.1: Proxy.huff#L129-L131 //Delegatethecalltotheimplementation.retOffsetandretSizeare
//0x00becausewedon'tknowthesizeyet. delegatecall// [success] Figure2.2: Proxy.huff#L148-L150
CallHelper.sol 's functionDelegateCallUnverified functionalso lackssuchacheck:
functionfunctionDelegateCallUnverified(addresstarget,bytesmemorydata) internal returns(bytesmemory)
{ (boolsuccess,bytesmemoryreturndata)=target.delegatecall(data); if(success){ returnreturndata; }
_revert(returndata); } Figure2.3: CallHelper.sol#L53-L62
TheSoliditydocumentationincludesthefollowingwarning: hashey 20NestedFinanceSecurityAssessment
PUBLIC

```

The low-level functions `call`, `delegatecall` and `staticcall` return `true` as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed. Figure 2.4: A snippet of the Solidity documentation detailing unexpected behavior related to `delegatecall`. As a result, any call made to a non-existent contract will return success, even if no code was executed. Note that function `DelegateCallUnverified`'s documentation states that this check is not needed:

```

//WedonotuseverifyCallResultFromTargetastheHyVMwill
//practicallyneverreturnanything.Inthecontextoftetris,weknow
//thatthetargetwillalwaysbeacontract.Hence,removingthis ///checkallowstoreducegasconsumption.

```

Figure 2.5: CallHelper.sol#L46-L49 However, the vulnerability highlighted in issue TOB-NESTED-1 indicates that this optimization is unsafe. Exploit Scenario

Bob uses a nested wallet for his portfolio. Bob has a \$10 million position that is close to being liquidated. Eve uses the vulnerability described in TOB-NESTED-1 to destroy the HyVM contract. Bob tries to add liquidity through the execution of a HyVM payload; his transactions succeed, but no code is executed. As a result, Bob's position is liquidated. Recommendations

Short term, implement a contract existence check before every low-level call.

Long term, carefully review the Solidity documentation, especially the "Warnings" section. Document every assumption made for code optimizations, and ensure that the underlying invariants hold. For example, if an invariant is that all the contracts that are used exist, extra care must be taken to ensure that all calls' destinations always have code.

hashey 21NestedFinanceSecurityAssessment PUBLIC

7. Risk of overflow in `FIX_MEMOFFSET` that could allow attackers to overwrite protected memory

Severity: High Difficulty: Medium Type: Data Validation Finding ID: TOB-NESTED-7 Target: Pointer.huff

Description A pointer overflow could occur in the `FIX_MEMOFFSET` macro, which could allow attackers to overwrite any memory slot, including the execution pointer, jump table, and the contract verifiers space. The HyVM reserves the first memory slots for internal use:

Figure 7.1: The "Memory Layout" section of the HyVM documentation hashey 22NestedFinanceSecurityAssessment PUBLIC

To ensure that the execution of a user's smart contract does not impact these memory slots, `FIX_MEMOFFSET` shifts any memory access by the size of the protected memory: `#definemacroFIX_MEMOFFSET()=takes(1)returns(1){ //Takes// [mem_offset] [HOST_MEMORY_START]add//[mem_offset+HOST_MEMORY_START] }`

Figure 7.2: Pointer.huff#L44-L47 However, there is no overflow protection on the `add` operation. As a result, an attacker can read/write to any protected memory by providing a large offset.

If an attacker can read/write to protected memory, they could overwrite the memory used by the verifier or control the execution pointer and jump to any `JUMPDEST`. In the latter case, the attacker could use this control to destroy the HyVM contract (similar to the issue described in finding TOB-NESTED-1). For example, if the verifier is enabled, the attacker could bypass all of the `delegatecall` protections by jumping to `contract-verifier.checkcall_success`. This is also possible if `op_delegatecall` is refactored to include a `JUMPDEST` after the checks. Exploit Scenario

The verifier is enabled. Eve calls the HyVM contract directly and causes an overflow in `FIX_MEMOFFSET`; the execution jumps directly over `contract-verifier.checkcall_success` and bypasses the `delegatecall` protections. Eve has the HyVM make a `delegatecall` to a contract that executes `selfdestruct`. As a result, Eve destroyed the HyVM contract. Recommendations

Short term, add an overflow check to `FIX_MEMOFFSET`. Long term, use a fuzzer to ensure that users cannot arbitrarily change the protected memory.

hashey 23NestedFinanceSecurityAssessment PUBLIC

8. Users can directly call the HyVM implementation Severity: Informational Difficulty: Low

Type: Undefined Behavior Finding ID: TOB-NESTED-8 Target: HyVM.huff Description Although the HyVM contract is intended to be called via `delegatecall` (so that the code is executed in the context of the caller), the contract also allows users to make direct calls to the implementation contract. This behavior is risky, and it is unclear whether this feature is needed. Because of the increased risk associated with this issue, extra care must be taken so that no forbidden op code is executed in the context of the HyVM.

The following are some of the risks associated with allowing users to call the HyVM contract directly: • These settings of storage values that the HyVM relies on (as anyone could set them) • The use of the HyVM as a swap router and to approve tokens spending • The risk that users could have the HyVM call an OFAC-sanctioned address, tainting the HyVM (and possibly all users) • The emission of inappropriate or deceiving events from the HyVM

**Recommendations** Short term, prevent the HyVM from being called directly; allow it to be called only via `delegatecall`. Long term, carefully identify the features that are needed for Nested wallets, and focus the development of the HyVM based on those requirements. `hashey` 24 Nested Finance Security Assessment PUBLIC

**9. State-changing operations permitted in HyVM** Severity: Informational Difficulty: Low  
Type: Undefined Behavior Finding ID: TOB-NESTED-9 Target: HyVM.huff Description The HyVM contract allows users to execute risky operations, including state-changing operations, which could make the use of the HyVM through Nested wallets risky and error-prone. Among other risky operations, HyVM allows users to execute the following: • `SSTORE` and `SLOAD` to read and write to a Nested wallet's storage • `delegatecall` and `callcode` to execute arbitrary code • `create` and `create2` to create new contracts

Although the HyVM was designed to be a generic VM, the need to include support for any Nested wallet use case is unclear. Management of a Nested wallet's storage layout through user-provided scripts is highly risky and is likely to lead to mistakes. This also significantly increases the attack surface, allowing attackers to more easily exploit users if they can trick them into executing malicious code. Moreover, the current version of the Nested SDK does not seem to use any state-changing operations, indicating that such operations are not required for using Nested wallets.

**Recommendations** Short term, disable `delegatecall`, `callcode`, and `SSTORE`. Consider also disabling `SLOAD`, `create`, and `create2`. Long term, evaluate the features required for the HyVM, and disable any unnecessary operations. Change the design paradigm to strive for simplicity over modularity and support for any use case. `hashey` 25 Nested Finance Security Assessment PUBLIC

**10. Lack of completed documentation on the known HyVM/EVM divergences** Severity: Informational Difficulty: Low  
Type: Undefined Behavior Finding ID: TOB-NESTED-10 Target: HyVM.huff Description The HyVM does not strictly follow the EVM specification. The known divergences are not clearly documented, which increases the likelihood that users will make mistakes while using the HyVM. The following are some of the HyVM's behaviors that do not align with the EVM: • `calldata.load`: The opcode's documentation in the HyVM contract reads, "getting `calldata` is not supported in `avm`, but instead of reverting, `calldata.load` puts the `mockcalldata` `0x0` on the stack without warning." • `calldata.copy`: This opcode has the same documentation as `calldata.load`, except it puts `0x0` in memory. • `calldata.size`: This opcode always puts a zero on the stack. • `delegatecall`: This opcode reverts (unless it is called in the context of a `delegatecall`). • `selfdestruct`: This opcode reverts. • `jumpdest`: This opcode does not include a check to ensure the validity of the `opcode`. • `codesize`: This opcode returns the `calldata.size` and not the VM size.

Some of these choices seem arbitrary, and users will have to go through the HyVM code to identify them.

**Recommendations** Short term, list all the expected divergences between the HyVM and EVM in a centralized document. Explain and justify every divergence. Long term, use differential fuzzing to identify additional divergences between the HyVM and EVM. `hashey` 26 Nested Finance Security Assessment PUBLIC

**11. Outdated HyVM used by nested-core-tetris** Severity: Informational Difficulty: Low  
Type: Undefined Behavior Finding ID: TOB-NESTED-11 Target: nested-core-tetris/lib/HyVM/src/HyVM.huff Description The nested-core-tetris folder uses an outdated version of HyVM. This version is error-prone and increases the likelihood that changes in HyVM will not be properly handled in the core contracts. As of this writing, the latest commit for HyVM, used during this code review, is `4e760d4535cf0e55d4453583c9b1484e5f3d3f9c`. However, nested-core-tetris is still using `849a1922f5f015bfe8f6f0089a31efe851d3665d`, which is a commit from October of 2022. Several changes have been made to HyVM since this commit:

Figure 11.1: The changes made between commits `4e760d` and `849a19`

The changes made significant modifications, including the following: • A refactor of the `jump` table • New opcodes • Changes in the opcode semantics (e.g., the use of the word "jump") • nested-core-tetris's use of the HyVM contract for testing in `NestedWallet.t.sol`, `WalletFactory.invariants.t.sol`, `WalletFactory.t.sol`, `LegoUseCases.t.sol`, and `NestedDca.t.sol` Moreover, running the tests with the latest HyVM version does not result in any failures, indicating a lack of in-depth testing of the HyVM integration in the core contracts. `hashey` 27 Nested Finance Security Assessment PUBLIC

**Exploit Scenario** Tests are rewritten that assert that the HyVM contract does not revert on `delegatecall` in certain situations, and new code is deployed based on this fallacy. Bob uses a Nested wallet for his portfolio. He has a \$10 million position that is close to being liquidated. Bob submits code to add collateral, but it reverts. As a result, Bob's position is liquidated.

**Recommendations** Short term, update nested-core-tetris's use of the outdated HyVM contract with the

latest version. Additionally, consistently use the latest dependencies across all repositories. Increase the integration tests of HyVM in the core contracts.

Long term, enable CI actions to ensure the correct dependencies are being used. Ensure that adequate testing is performed to catch potential differences. Consider using mutation testing to increase the confidence in the test suite (see [necessist's Solidity support](#)).  
hashey 28 NestedFinanceSecurityAssessment PUBLIC

12. Risks associated with using the Huff programming language Severity: Informational Difficulty: Low  
Type: Patching Finding ID: TOB-NESTED-12 Target: HyVM.huff , proxy.huff Description

The use of the Huff programming language leads to risks related to its unstable code that are not justified and maybe unnecessary. The Huff programming language is still under heavy development, and there are no stable releases of it yet. NestedFinance uses a specific unidentified nightly version of the Huff compiler in its GitHub CI tests in order to compile HyVM ; otherwise, recent compiler versions will crash (e.g., with nightly version e0f640f12373e0a98edfc1e33eb4cc40c6bec129 ).  
\$ huffc src/HyVM.huff -c Compiling... thread 'main' has overflowed its stack fatal runtime error: stack overflow Aborted  
Figure 12.1: A crash that occurs when trying to compile HyVM without the specific version of the Huff compiler used by NestedFinance. Additionally, to address a bug in the Huff compiler that causes the built-in Huff `__codesize` function to return an incorrect amount when called from within the constructor itself, the code base contains a workaround that is specific to the Huff version used: in the constructor for HyVM.huff , the return value of `__codesize` is adjusted by adding two (bytes). When a new version of the Huff compiler containing a fix for this bug is released, if HyVM continues to use the latest compiler version, then the runtime code will be two bytes short, overwriting the last two bytes of the code and left-shifting the address by two bytes.  
/// @dev The constructor stores the address of the HyVM at the end of the runtime bytecode. See `op_delegatecall`.  
#define macro CONSTRUCTOR() = takes(0) returns(0) { // store address of HyVM contract at the end of the runtime code  
// memory position at construction time. // It seems `__codesize(CONSTRUCTOR)` does not take into account 2 bytes  
// hence we fix it manually  
hashey 29 NestedFinanceSecurityAssessment PUBLIC

0x02 // [0x02] `__codesize(CONSTRUCTOR)` // [`__codesize(CONSTRUCTOR)`, 0x02] `add` // [offset] Figure 12.2:

HyVM.huff#L22-L31 Exploit Scenario The known Huff compiler bug mentioned above is fixed, and HyVM is directly deployed. Because HyVM is still adding two bytes to the code size, the layout of the bytecode is incorrect, and the retrieval of HyVM 's address in the code section leads to an incorrect value. As a result, the `delegatecall` protection does not work. Even makes HyVM directly execute a `delegatecall` to a contract that runs `selfdestruct` , which destroys the HyVM contract. Recommendations  
Short term, properly document the version of the Huff compiler and all of the workarounds related to it that are used in the code bases. Create reference implementations in Solidity for every contract written in Huff and compare their execution to the execution of the Huff contracts. Long term, carefully review every design choice and always strive for simplicity. Make sure to justify every use of a new programming language.  
hashey 30 NestedFinanceSecurityAssessment PUBLIC

Summary of Findings: TetrisCore

The table below summarizes the findings of the review, including type and severity details. ID Title Type Severity  
13 TimeLockControllerEmergency could be used to allow anyone to execute proposals Configuration Low  
14 Risks associated with the use and design of the Huff proxy Undefined Behavior Informational  
15 Use of `tx.origin` for access controls Access Controls Medium  
16 Ability to create wallets before `WalletFactory` is initialized Data Validation Low  
hashey 31 NestedFinanceSecurityAssessment PUBLIC

Detailed Findings: TetrisCore 13. TimeLockControllerEmergency could be used to allow anyone to execute proposals Severity: Low Difficulty: High Type: Configuration Finding ID: TOB-NESTED-13 Target: TimeLockControllerEmergency.sol Description

The `TimeLockContract` can be configured to use open roles, which allow any user to run proposals. The `TimeLockControllerEmergency` contract is implemented using `OpenZeppelin's TimeLockController` contract to add a delay for governance decisions to be executed:  
/// @author NestedFi  
/// @author Extends TimeLockController from 0Z  
/// @notice It introduces the "Emergency" role, which bypasses the time lock  
/// process. This role should be a multisig with more members than the  
/// operational multisig, and can be used in cases where the time lock  
/// delay is a problem (e.g. in the case of a critical vulnerability // that needs to be fixed immediately).  
contract TimeLockControllerEmergency is TimeLockController { ...  
Figure 13.1: The definition of the `TimeLockControllerEmergency` contract in `TimeLockControllerEmergency.sol`  
The `OpenZeppelin` code implements two functions to execute proposals using "open roles":  
// This function can reenter, but it doesn't pose a risk because `_afterCall` checks that the proposal is pending,  
// thus any modifications to the operation during reentrancy should be caught. // `slither-disable-next-line`  
`reentrancy-eth` function `execute( address target, uint256 value, bytes callData payload,`

```
bytes32predecessor, bytes32salt )publicpayablevirtualonlyRoleOrOpenRole(EXECUTOR_ROLE){ hashey  
32NestedFinanceSecurityAssessment PUBLIC
```

```
... } functionexecuteBatch( address[]calldatatargets, uint256[]calldatavalues,  
bytes[]calldatapayloads, bytes32predecessor, bytes32salt  
)publicpayablevirtualonlyRoleOrOpenRole(EXECUTOR_ROLE){ ... } Figure13.2:Partofthe  
TimeLockController.sol codefromOpenZeppelin
```

Theuseofopenrolesmeansthatifthezeroaddressisaddedwiththeexecutorrole,then anyonewillbeabletoexecuteproposals.Ifopenrolesisafeaturethatshouldneverbe used,thenitshouldbedisallowedwhenthe TimeLockControllerEmergency contractis deployed. However,thecontract'sconstructordoesnotcontaincodetopreventtheuseofthezero addressand,therefore,openroles: constructor( uint256minDelay, address[]memoryproposers, address[]memoryexecutors, addressemergency )TimeLockController(minDelay,proposers,executors){ //Adminroleforemergency \_setRoleAdmin(EMERGENCY\_ROLE,TIMELOCK\_ADMIN\_ROLE); //registeremergency \_setupRole(EMERGENCY\_ROLE,emergency); } Figure13.3:Theconstructorofthe TimeLockControllerEmergency contract ExploitScenario Alicecreatesa TimeLockControllerEmergency contract,whichismisconfiguredusing onezeroaddress.Asaresult,Eveisallowedtorunproposals. Recommendations Shortterm,eitherproperlydocumenttheopenrolesfeatureordisallowtheuseofzero addresseswhensettingroles. hashey 33NestedFinanceSecurityAssessment PUBLIC

Longterm,carefullyreviewanddocumenttheassumptionsandlimitationsregarding third-partycodeintegrationsandconsiderwhetherthelimitationsareacceptableand whethertheassumptionshold. hashey 34NestedFinanceSecurityAssessment PUBLIC

14.RisksassociatedwiththeuseanddesignoftheHuffproxy Severity:InformationalDifficulty:Low Type:UndefinedBehaviorFindingID:TOB-NESTED-14 Target: Proxy.huff Description ThedesignanduseoftheHuffproxycontractintroducerisksthatarenotjustifiedandmay beunnecessary. TheHuffproxyisusedintwoplaces: • Thewalletfactorycontract • TheNestedwalletcontract TheuseofHufftowritetheproxyisjustified:toreducetheproxy'ssize.However,theuse oftheproxyinthewalletfactoryisunlikelytomeaningfullyreducethecostofdeploying thecontract. Moreover,theproxyretrievestheimplementation'saddressusingaresolverwithdynamic dispatching: hashey 35NestedFinanceSecurityAssessment PUBLIC

Figure14.1:The“Mechanism”sectionofthe nested-core-tetris documentation Here,theproxycallstheimplementationresolverwitheither“ WalletFactory” or “ NestedWallet” . Whentheimplementationisconstructed,theproxyappendsthennameofthe implementationtotheendofthebytecode.Atruntime,theproxythenretrievesthename fromthebytecodebycopyingitsvalue. AlloftheseoperationsfollowthesameschemathatSolidity'simmutablevariablesdo,but theyaredonethroughrawopcode manipulations: /\*-----  
-----\*/ /\*MACROS\*/ /\*-----  
-----\*/ ///@devCopytheimmutableimplementationnametomemory. /// hashey  
36NestedFinanceSecurityAssessment PUBLIC

```
///Immutableareappendedattheendoftheruntimebytecode likethe ///constants.  
#definemacroCOPY_TO_MEMORY_IMPLEMENTATION_NAME()=takes(0)returns(0){  
[IMPLEMENTATION_NAME_SIZE]//[size] dup1codesizesub//[offset,size]  
[IMPLEMENTATION_NAME_OFFSET]//[destOffset,offset,size] codecopy//[ ] } Figure14.2: Proxy.huff#L89-  
L95 Thispatternishighlyerror-prone.Becausealloftheseoperationscouldbereplacedwith  
oneimplementationresolverpertypeofcontract,therisksthatcomewithusingthis patternmaybeunnecessary.  
Recommendations Shortterm,useaSolidityproxyforthefactorycontract.Useoneimplementationresolver  
pertypeofcontract. Longterm,carefullyrevieweverydesignchoiceandalwaysstriveforsimplicity.Makesure  
tojustifyeveryuseofassemblycode. hashey 37NestedFinanceSecurityAssessment PUBLIC
```

15.Useoftx.originforaccesscontrols Severity:MediumDifficulty:High Type:AccessControlsFindingID:TOB-NESTED-15 Target: WalletFactory.sol Description Thewalletfactoryuses tx.origin toauthenticateusers.Thispatterniscommonly consideredinsecure,anditcouldresultinthecreationofundesiredwalletsorwalletsina vulnerablestate. WalletFactory allowsuserstocallthe createAndCallTx0origin functiontocomplete walletswhoseowneris tx.origin : functioncreateAndCallTx0origin(bytes32salt,bytescalldatacall) external payable override nonReentrant returns(addressnestedWallet,bytesmemorydata) { addresssender=tx.origin; //DeployaNestedWallet nestedWallet=\_deployWallet(salt,sender); //TheWalletFactorymustbetemporarilytheowntorto beabletoforward //thecalltotheWallet. owners[nestedWallet]=address(this); //CallthedevelopedNestedWallet data=CallHelper.functionCall(nestedWallet,call,msg.value); // (Re)setthesenderasownerofthecreatedwallet owners[nestedWallet]=sender; emitNestedWalletCreated(nestedWallet,sender); } Figure15.1: WalletFactory.sol#L175-L198

However, the user creating the transaction might not be aware that this function is executed, which would create an undesired wallet. Moreover, a malicious initial call to the `hashey` 38NestedFinanceSecurityAssessment PUBLIC

`CallHelper.functionCall` is executed in `createAndCallTxOrigin`, which could put the created wallet in a vulnerable state. `ExploitScenario`  
Bob frequently creates nested wallets that hold USDC tokens. Even notice this and tricks Bob into transferring a token with a callback, and she uses the callback to call `createAndCallTxOrigin`. During the wallet's initialization, the wallet approves `Eveto` transfer all of its USDC tokens. The new wallet is shown in the Nested Finance interface. Bob starts using the wallet. After some time, the wallet holds \$10 million worth of USDC. `Evestealsthetokens.`  
Recommendations Short term, remove `createAndCallTxOrigin`. Long term, never use `tx.origin` for authentication. `hashey` 39NestedFinanceSecurityAssessment PUBLIC

16. Ability to create wallets before `WalletFactory` is initialized Severity: Low Difficulty: High  
Type: Data Validation Finding ID: TOB-NESTED-16 Target: `WalletFactory.sol` Description  
When a user creates a new wallet through the `WalletFactory` contract, a check for whether the factory has been initialized correctly is missing. This missing check can result in the creation of unusable wallets. When a new wallet is created, the `_deployWallet` function is called, which deploys a proxy contract given the bytecode from the `_bytecodeWithArgs` function.  
/// @dev Return the bytecode with args necessary for create2.  
/// It is not stored in storage as it is more expensive than using /// memory.  
function `_bytecodeWithArgs`() private view returns (bytes memory) {  
return abi.encodePacked(proxyByteCode, implementationResolver, bytes32("NestedWallet")); }  
Figure 16.1: The `_bytecodeWithArgs` function in `WalletFactory.sol` If the `_deployWallet` function is called but the `WalletFactory` contract has not been yet initialized, then the `proxyByteCode` variable will be empty. And if the bytecode resulting from the `abi.encodePacked(implementationResolver, bytes32("NestedWallet"))` function can be interpreted as a valid executable bytecode, then the wallet creation will succeed, creating a non-functioning wallet. The above situation can happen when the implementation resolver address starts with a certain value, such as `0x00`, which is interpreted as the `stop` opcode.  
As noted by Nested Finance, this issue is currently not reachable due to the use of the reentrancy modifier. See the TOB-NESTED-16 fix review results in the fix review appendix for more details. `ExploitScenario`  
Alice creates a new `NestedWallet` proxy. The contract creation succeeds; however, Alice is unable to use her wallet because the contract deployer forgot to initialize `WalletFactory`. `hashey` 40NestedFinanceSecurityAssessment PUBLIC

Recommendations Short term, have `_deployWallet` check whether `initialized` is true or require that the proxy bytecode be non-empty.  
Long term, create diagrams or flowcharts on the wallet creation process, including the creation of `WalletFactory`. Document any requirements to be fulfilled for any function, and ensure that the relevant functions revert if those conditions are not met. `hashey` 41NestedFinanceSecurityAssessment PUBLIC

#### Summary of Findings: Token

The table below summarizes the findings of the review, including type and severity details. ID Title Type Severity  
17 Risk of portfolio voting power theft Data Validation High  
18 acceptDeal adds unreleased tokens to new vesting schedules Data Validation Low  
19 Vesting contract owner is not updated when VestingFactory owner changes Authentication Informational  
20 Unbounded loops could cause denial of service for third-party integrations Data Validation Low `hashey` 42NestedFinanceSecurityAssessment PUBLIC

Detailed Findings: Token 17. Risk of portfolio voting power theft Severity: High Difficulty: Low  
Type: Data Validation Finding ID: TOB-NESTED-17 Target: `Bribes.sol` Description  
Due to incorrect bookkeeping, anyone can steal a portfolio's voting power. The `deposit` function deposits the given number of the user's `nst(amount)` and increases the chosen portfolio's power by that number.  
function `deposit`(address ptf, uint256 amount) external override nonReentrant {  
if (amount == 0) revert InvalidAmount(amount); powers[ptf] += amount; deposits[\_msgSender()] += amount;  
nst.safeTransferFrom(\_msgSender(), address(this), amount); emit Deposited(ptf, \_msgSender(), amount); }  
Figure 17.1: The `deposit` function in `Bribes.sol` The `withdraw` function does the opposite—it withdraws the user's deposit and decreases the chosen portfolio's power by the number of tokens withdrawn.  
function `withdraw`(address ptf, uint256 amount) external override nonReentrant {  
if (amount == 0) revert InvalidAmount(amount); if (deposits[\_msgSender()] < amount) revert AmountExceedsDeposit(amount); powers[ptf] -= amount; deposits[\_msgSender()] -= amount;  
nst.safeTransfer(\_msgSender(), amount); emit Withdrawn(ptf, \_msgSender(), amount); }  
Figure 17.2: The `withdraw` function in `Bribes.sol` `hashey` 43NestedFinanceSecurityAssessment PUBLIC

However, there is a check to ensure that the portfolio is closed with draw as the one used to deposit. As a result, an attacker could deposit NST to increase the power of portfolio A and withdraw NST to decrease the power of portfolio B. **Exploit Scenario** Bob delegates 100 NST of bribes to Alice's portfolio. Eve allocates 100 NST of bribes to herself. Eve then removes 100 NST of bribes from Alice's portfolio. As a result, Eve has stolen 100 NST of voting power from Alice's portfolio. **Recommendations** Short term, use mapping(address depositor => address portfolio) to track portfolio bookkeeping. Long term, carefully document the data structure choices and their assumptions. Ensure that proper validation and tests are replaced on operations that impact and are impacted by user funds. Consider using Echidna to test multiple-transaction invariants on the Nested contracts. **hashey** 44NestedFinanceSecurityAssessment PUBLIC

18. **acceptDeal** adds unreleased tokens to new vesting schedules **Severity: Low Difficulty: Medium**  
**Type: Data Validation Finding ID: TOB-NESTED-18 Target: Vesting.sol Description** Claimable tokens become unclaimable if new vesting schedules are recreated, delaying the opportunity for users to withdraw. When a deal is accepted through the **acceptDeal** function during an active vesting schedule, a new vesting schedule is automatically pushed. **function acceptDeal(address recipient, uint256 amount) external override onlyOwner onlyTeam{ ... // Create a new schedule, if it's after the cliff and not fully vested if( block.timestamp > currentSchedule.start && block.timestamp < currentSchedule.start + currentSchedule.duration ){ schedules.push( Schedule({ start: block.timestamp, duration: currentSchedule.duration - (block.timestamp - currentSchedule.start), released: 0, dealt: 0 } ) ); } // Remove the current deal currentDeal = bytes32(0); emit DealAccepted(recipient, amount); }** **Figure 18.1: The acceptDeal function in Vesting.sol** However, claimable tokens in the previous schedule will be added to the new vesting schedule instead of being released. This creates a delayed vesting schedule and a slower token release than expected. **hashey** 45NestedFinanceSecurityAssessment PUBLIC

**Exploit Scenario** Bob receives 1,001 NST that will be vested over the next two years. After one year, Bob creates an OTC deal for 1 NST. Bob still has not claimed any of his claimable NST tokens (around 500 NST). Alice, the owner, accepts Bob's deal, and Bob receives 1 vested NST. Because his deal was accepted, a new vesting schedule is automatically created for Bob. The new schedule is set to release all of his remaining 1,000 NST over the remaining year. Now Bob is temporarily unable to claim his originally claimable 500 NST. **Recommendations** Short term, include a call to **\_releaseNst** in **acceptDeal** so that all claimable tokens are released before a new vesting schedule is pushed. Long term, create a diagram highlighting the state transitions of the **Vesting** contract. In this diagram, emphasize the transition to new vesting schedules and identify underlying invariants. Ensure that unit tests cover every transition. Consider using Echidna to test invariants related to vesting. **hashey** 46NestedFinanceSecurityAssessment PUBLIC

19. **Vesting contract owner is not updated when VestingFactory owner changes** **Severity: Informational Difficulty: High Type: Authentication Finding ID: TOB-NESTED-19 Target: VestingFactory.sol Description** After ownership of the **VestingFactory** contract is transferred, the previous owner still controls the previously created **Vesting** contracts and can withdraw funds from them. It is unclear whether this behavior is intended. When creating a **Vesting** contract, **VestingFactory** assigns its own current owner to the created **Vesting** contract's owner: **function createVesting( bool teamBeneficiary, address beneficiary, uint256 cliff, uint256 duration, uint256 leaverIncrease, uint256 allocation ) external onlyOwner returns( Vesting newVesting ){ newVesting = newVesting( nst, teamBeneficiary, beneficiary, owner(), cliff, duration, leaverIncrease ); vestings.push( newVesting ); IERC20( nst ).safeTransfer( address( newVesting ), allocation ); emit VestingCreated( address( newVesting ), allocation ); }** **Figure 19.1: The createVesting function in VestingFactory.sol** Ownership of the **VestingFactory** contract can be transferred in a two-step process. If the **VestingFactory** owner changes, all new **Vesting** contracts will be owned by the new **VestingFactory** owner; however, the previous **VestingFactory** owner will continue to control the existing **Vesting** contracts. As a result, two different owners will control different sets of **Vesting** contracts. The previous owner can withdraw funds from **Vesting** contracts before their vesting schedules complete. **Recommendations** Short term, document the process so that users are informed of who will control a **Vesting** contract when the **VestingFactory** owner is transferred. **hashey** 47NestedFinanceSecurityAssessment PUBLIC

Long term, exercise caution when implementing code that reuses contract owners for other contracts; ensure that transactions for transferring ownership of all the contracts owned by a single address are executed at once. **hashey** 48NestedFinanceSecurityAssessment PUBLIC

20. **Unbounded loops could cause denial of service for third-party integrations** **Severity: Low Difficulty: High Type: Data Validation Finding ID: TOB-NESTED-20 Target: VestingFactory.sol, Vesting.sol Description** Unbounded loops to compute the total number of tokens released in the **VestingFactory** and **Vesting**

contracts can result in denial of service if the contracts integrate with these Vesting contracts. The VestingFactory contract creates Vesting contracts and stores their addresses in the vestings variable. The totalReleased function in the VestingFactory contract iterates over this list of vesting contracts and calls itself on the given Vesting contract to compute the total number of tokens released to all of the beneficiaries.

```
function totalReleased() external view returns (uint256 total) { for (uint256 i; i < vestings.length; i++) { total += vestings[i].totalReleased(); } }
```

Figure 20.1: The totalReleased function in VestingFactory.sol

The totalReleased function in the Vesting contract also iterates over the list of schedules to compute the total number of tokens released to the beneficiaries. A new schedule is added to the list for every deal made between the start and end of the vesting period. This way, the number of schedules can grow to any arbitrary number.

```
function totalReleased() external view override returns (uint256) { uint256 total; for (uint256 i = 0; schedules.length > i; i++) { total += schedules[i].released; } return total; }
```

Figure 20.2: The totalReleased function in Vesting.sol

Therefore, the totalReleased function in the VestingFactory contract can throw an out-of-gas error if too many vestings or schedules are added, resulting in denial of service of the contract calling this function. [hashey 49 Nested Finance Security Assessment PUBLIC](#)

Similar unbounded loops are used in other functions like totalReleasable and totalAllocation in the VestingFactory contract and the totalDealed function in the Vesting contract. All of these functions can also cause denial of service of the contract calling them in a transaction.

**Exploit Scenario** The Nested Finance team creates 100 Vesting contracts, one for each team member and stakeholder. Eve, one of the team members, creates 100,000 deals to sell her tokens during the vesting period. The owner accepts all of these deals. Now, any contract calling the totalReleased function on the VestingFactory contract will need to iterate through 10,000,000 items, which may result in an out-of-gas error and make the contract unusable.

**Recommendations**

**Short term**, document this behavior to make sure third parties are aware of the risks.

**Long term**, carefully review operations that consume a large amount of gas, especially those in loops. Model any variable-length loop to ensure that they cannot block contract execution within the expected system parameter limits. [hashey 50 Nested Finance Security Assessment PUBLIC](#)

#### Summary of Findings: DCA

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

ID	Title	Type	Severity
21	DCA ownership transfers can be abused to steal tokens	Access Controls	High
22	Risk of unlimited slippage in Nested DCA swaps	Data Validation	High
23	Insufficient DCA existence check could result in an inconsistent state	Data Validation	Low
24	Users prevented from using special ETH address to pay Gelato fees	Data Validation	Low
25	DCA tasks can be stopped by anyone	Data Validation	Medium
26	DCA task ID can be manually set	Data Validation	Low
27	restartDca lacks data validation	Data Validation	Medium
28	Reentrancy vulnerabilities in Nested DCA could allow Gelato operators to drain left over tokens	Data Validation	Low
29	Unclear usage of Dca.tokensOutAllocation	Undefined Behavior	Informational

[hashey 51 Nested Finance Security Assessment PUBLIC](#)

**Detailed Findings: DCA**

21. DCA ownership transfers can be abused to steal tokens. Severity: High Difficulty: Low  
Type: Access Controls Finding ID: TOB-NESTED-21 Target: NestedDca.sol Description  
DCA ownership is transferred without the consent of the new owner, which means that DCA ownership transfers can be abused to steal users' tokens.

The recipient of a DCA swap is stored in the swap parameter `ISwapRouter.ExactInputSingleParams.recipient`.

```
struct Dca { uint120 lastExecuted; uint112 interval; uint16 swapSlippage; bool isGelatoWatching; uint16[] tokensOutAllocation; bytes32 taskId; ISwapRouter.ExactInputSingleParams[] swapsParams; }
```

Figure 21.1: The Dca struct in INestedDca.sol

In most cases, the recipient of the swap will also be the creator of the DCA, stored in `ownerOf[dcaId]`. When an automated swap is performed, the tokens are transferred from the current owner of the given DCA to the recipient defined in the swap parameters.

```
IERC20(swapParams.tokenIn).safeTransferFrom(ownerOf[dcaId], address(this), swapParams.amountIn ); // Perform the DCA programmed swap amountsOut[i] = _performDcaSwap(dcaId, i);
```

Figure 21.2: The performDca function in NestedDca.sol

When ownership of a DCA is transferred via the `transferDcaOwnership` function, the function sets the new owner without checking for the new owner's consent. If ownership is [hashey 52 Nested Finance Security Assessment PUBLIC](#)

transferred after the DCA has already been set up and started, then the tokens will be transferred from the new user instead.

This issue can be abused to steal tokens from any user that has already given their token-spending approval to the NestedDca contract.

**Exploit Scenario**

Bob sets up a DCA to convert this USDC token to ETH. For this purpose, he has given the NestedDca contract unlimited approval. He currently has \$1 million worth of USDC in his

wallet.Eventhispoint,thepersonwhoowns\$1millionofUSDC(thatsheownsnotown)toETHinasingleswap.ShethenstartstheDCAandimmediatelytransferstheownershipoftheDCAtoBob.OnceEve'sautomatedtaskisperformed,Bob's\$1millionofUSDCistransferredtoEve.

Recommendations Shortterm,eitherdisallowtransfersofDCAownershiporaddatwo-stepownershiptransferprocess,inwhichthenewownermustaccepttheownershiptransfer.

Longterm,createdocumentation/diagramsthathighlightallofthetransfersoffasetsandthetokenapprovals.Addadditionaltests,anduseEchidnatotestsysteminvariants targetingtheassetsandDCAtransfers. hashey 53NestedFinanceSecurityAssessment PUBLIC

22.RiskofunlimitedslippageinNestedDcaswaps Severity:HighDifficulty:Low  
Type:DataValidationFindingID:TOB-NESTED-22 Target: NestedDca.sol Description TheswapsexecutedbytheNestedDca contractfromUniswapV3allowunlimitedslippage becausetheslippageprotectionisassignedtoamemoryvariablethatisnotpassedtothenextfunctionexecutingtheswap.

WhenexecutingswapsonUniswap,aslippagecheckparameterisrequiredtoprotectusersfrompayinganunexpectedlyhigheramountforthetokenbeingbought. The NestedDca contractexecutestheDCAbyswappinguser-configuredtokensinaloop, asshownbelow:

```
for(uint256i;i<swapsAmount;){ ISwapRouter.ExactInputSingleParamsmemoryswapParams=dcas[dcaId].swapsParams[i]; addressuniswapPool=uniswapFactory.getPool(swapParams.tokenIn,swapParams.tokenOut, swapParams.fee);if(pools[uniswapPool]==0)revertUnauthorizedUniswapPool(uniswapPool);//ComputetheminimumacceptableoutputamountswapParams.amountOutMinimum=ExchangeHelpers.estimateDcaSwapAmountOut(uniswapPool,swapParams,uint32(pools[uniswapPool]), dcas[dcaId].swapSlippage );IERC20(swapParams.tokenIn).safeTransferFrom( ownerOf[dcaId],address(this),swapParams.amountIn );//PerformtheDCAprogrammedswap amountsOut[i]=_performDcaSwap(dcaId,i); unchecked{ ++i; } } hashey 54NestedFinanceSecurityAssessment PUBLIC
```

Figure22.1:The performDca functionin NestedDca.sol

Theswapparametersareassignedfromcontractstoragetoalocalmemoryvariable named swapParams .Theslippagevalueiscomputedandsettothememoryvariable swapParams.amountOutMinimum .Thisvalueisnotsettothestoragevariable;therefore,itisvisibleonlyinthescopeofthisfunction,notinotherfunctionsthatreadswap parametersfromstorage. The performDca functionthencallsthe \_performDcaSwap function,whichagainassigns swapparametersfromcontractstoragetoalocalmemoryvariablenamed swapParams ,as shownbelow:

```
function_performDcaSwap(bytes32dcaId,uint256swapParamsIndex) private returns(uint256amountOut) {ISwapRouter.ExactInputSingleParamsmemoryswapParams= dcas[dcaId].swapsParams[swapParamsIndex];amountOut=uniswapRouter.exactInputSingle(swapParams); } Figure22.2:The _performDcaSwap functioninNestedDca.sol Sothevalueof swapParams.amountOutMinimum isnotcorrectinthescopeofthe _performDcaSwap function—it is 0 ,indicatingnoslippageprotection.Thisincorrectvalue isthensentasanargumenttotheUniswapV3router,whichexecutestheswapwithoutslippageprotection. Anattacker canfront-runorsandwichtheswaptransactiontobenefitfromthisunprotectedswaptransaction. ExploitScenario

AliceusesaDCAstrategytobuy$50,000worthofMyTokeneveryday.TheDCAisexecuted everydayat9a.m.ETbytheGelatooperators. Evefront-runsthetransaction,buys$10,000worthofMyToken,andsellsthemdirectlyafterAlice'stransaction. Asaresult,AliceboughtMyTokenatanunfairprice,andEveprofitedfromit. Recommendations



Shortterm,havethecodepassthevalueofthememoryvariable swapParams fromthe performDca functiontothe _performDcaSwap functionasanargument.



Longterm,carefullyreviewthecodetocheckforthe correctuseofvaluesettomemory variables. Addunittestcases thatmimicfront-runningtocapturesuchissues. hashey 55NestedFinanceSecurityAssessment PUBLIC


```

23.InsufficientDCAexistencecheckcouldresultinaninconsistentstate Severity:LowDifficulty:Low  
Type:DataValidationFindingID:TOB-NESTED-23 Target: NestedDca.sol Description AfterrenouncingownershipofaDCA, ausercanrecreatethesameDCAandloseaccesstothetasksassociatedwiththepreviousDCA. ThiscanbreakassumptionsmadebyGelato's componentsandbreakotherthird-partyintegrations. The createDca functioncomputesanIDforaDCAbyhashingtheparametersprovidedbytheuser. ItthenchecksfortheexistenceofaDCAwiththesameIDbycheckingwhetheranownerisalreadyassignedtothegiven dcaId .

```
functioncreateDca(Dcacalldata_dca)publicnonReentrantreturns(bytes32dcaId){if(_dca.isGelatoWatching)revertDcaMustNotBeStartedAtCreation(); //SetuptheDCAdcaId=_setupDca(_dca); if(ownerOf[dcaId]!=address(0))revertDcaAlreadyExist(dcaId); ...
```

Figure23.1:The createDca functionin NestedDca.sol The renounceDcaOwnership functionallowsauser torenounceownershipofaDCAby settingtheowner toaddress 0x0

. However, this function does not check whether the Gelato task created for the DCA has been stopped. `function renounceDcaOwnership(bytes32 dcaId) public virtual only DcaOwner(dcaId) { _transferDcaOwnership(dcaId, address(0)); }` Figure 23.2: The `renounceDcaOwnership` function in `OwnableDca.sol`. If a user who renounced ownership creates a new DCA with the same parameters, then the new DCA will have the same `dcaId` as the previous one and will reset the `isGelatoWatching` and `taskId` variables. Previously created tasks will no longer be associated with the DCA in the nested contract, preventing them from being stopped. They will still be enabled in the Gelato network. `Asa` `hashey` 56NestedFinanceSecurityAssessment PUBLIC

result, multiple unstopable tasks can be created for the same DCA. This could break the system's integrations with third-party components. **Exploit Scenario**  
Alice creates and starts a DCA. After some time, she renounces her ownership of it. Later, Alice creates a new DCA with exactly the same parameters. A new DCA gets created with her previous DCA's `dcaId`, which already has a Gelato task running for it. Alice starts the new DCA again and selects a different `feeToken`, allowing her to create a new Gelato task for her DCA, which will then set a new `dca.taskId` in the contract storage. This leaves Alice with two active Gelato tasks for one `dcaId`. **Recommendations**  
Short term, implement our recommendations for an alternative schema for storing DCAs in the `NestedDca` contract, proposed in appendix C.  
Long term, document the expected validation that should occur when a DCA is created. Create a diagram highlighting the lifecycle of a DCA and the underlying invariants. Create unit tests for each state transition, and consider using Echidna to test multiple-transaction invariants. `hashey` 57NestedFinanceSecurityAssessment PUBLIC

24. Users prevented from using special ETH address to pay Gelato fees **Severity: Low Difficulty: Low**  
**Type: Data Validation Finding ID: TOB-NESTED-24 Target: NestedDca.sol Description** The `startDca` function's check of the provided `feeToken` address blocks users from using ETH to pay Gelato fees. The `startDca` function allows users to start their DCA using a particular `feeToken` address to pay Gelato fees. As part of the validation, it checks that the `feeToken` address points to a deployed address (i.e., an address with deployed code).

```
function startDca(bytes32 dcaId, address feeToken) public override only DcaOwner(dcaId)
returns(bytes32 taskId) {
    DcaStorage dca = dcas[dcaId];
    if(dca.isGelatoWatching) revert DcaAlreadyStarted(dcaId);
    if(feeToken.code.length == 0) revert AddressNotContract(feeToken); ...
}
Figure 24.1: The startDca function in NestedDca.sol. Gelato fees are paid by the _transferGelatoFees function in the tokens specified by the user:
address internal constant ETH = 0xEeeeeEeeeEeEeeEeEeEeeeeEeeeeeeEEeE; ...
function _transferGelatoFees(bytes32 dcaId, uint256 feeAmount, address feeToken) private {
    if(feeToken != ETH) { IERC20(feeToken).safeTransferFrom(ownerOf[dcaId], gelato, feeAmount); } else {
        WETH.transferFrom(ownerOf[dcaId], address(this), feeAmount); WETH.withdraw(feeAmount);
        (bool success, ) = gelato.call{value: feeAmount}("");
        if(!success) revert EthTransferFail(address(this), gelato);
    }
}
hashey 58NestedFinanceSecurityAssessment PUBLIC
```

} } Figure 24.2: The `_transferGelatoFees` function in `NestedDca.sol`. However, the special ETH address (`0xEee..`) will never have deployed code; therefore, it cannot be used as a `feeToken`. If `0xEee` is supplied, the `else` branch in `_transferGelatoFees` is unreachable. **Exploit Scenario**  
Alice creates a DCA and wants to pay the Gelato fees using ETH. She checks the Gelato documentation and uses the correct `feeToken` address, but the `NestedDca` contract reverts when she tries to start the DCA. She is forced to use another token to pay her fees. **Recommendations**  
Short term, properly document the special ETH address and modify the `startDca` function to allow users to use the ETH address to pay Gelato fees.  
Long term, make sure that the code has enough unit and integration tests to test all of its features. `hashey` 59NestedFinanceSecurityAssessment PUBLIC

25. DCA tasks can be stopped by anyone **Severity: Medium Difficulty: Low Type: Data Validation Finding ID: TOB-NESTED-25 Target: NestedDca.sol Description** Stopping an active DCA task relies on a user-provided parameter, the `taskId`. However, there is no validation performed on this value, so any automated DCA task can be stopped by anyone. The `stopDca` function takes as input a DCA ID and a `gelato-ops` `taskId`.

```
function stopDca(bytes32 dcaId, bytes32 _taskId) public override only DcaOwner(dcaId) {
    DcaStorage dca = dcas[dcaId];
    if(!dca.isGelatoWatching) revert DcaAlreadyStopped(dcaId);
    IOps(ops).cancelTask(_taskId);
    dca.isGelatoWatching = false;
    emit DcaStopped(dcaId, _taskId);
}
Figure 25.1: The stopDca function in NestedDca.sol. Ownership of the DCA is verified through the onlyDcaOwner(dcaId) modifier. However, there are no checks performed on the _taskId parameter to verify that the given taskId belongs to the provided DCA ID (i.e., require(_taskId == dcas[dcaId].taskId)), or
```

that the given DCA has been successfully started at task in the first place. This means that anyone is able to supply a task ID of a running DCA to shut down another user's automated task. Exploit Scenario Alice sets up a task to dollar-cost-average her 100 million Token A back to USD cover the next month. Eve stops Alice's DCA task. After one month, Token A's value suddenly declines to nearly zero. Without her knowledge, Alice's token has lost all of its value, since the DCA task was not performed after being stopped by Eve without Alice's approval. hashey 60 Nested Finance Security Assessment PUBLIC

Recommendations Short term, add a check that verifies that the task ID matches the task ID from the provided DCA, or have the codes simply directly access the task ID stored for the given DCA. Long term, document the expected validation that should occur when a DCA task is stopped. Create a diagram highlighting the lifecycle of a DCA and the underlying invariants. Create unit tests for each state transition, and consider using Echidna to test multiple-transaction invariants. hashey 61 Nested Finance Security Assessment PUBLIC

26. DCA task ID can be manually set Severity: Low Difficulty: Low Type: Data Validation Finding ID: TOB-NESTED-26 Target: NestedDca.sol Description The task ID

parameter of a DCA should be set only by Gelato, but it can be arbitrarily set by the user. This can break assumptions made by Gelato's components and break other third-party integrations. The createDca function does not check whether the user has manually set the dca.taskId parameter, even though it verifies that other parameters have not been manually set.

```
function createDca(DcaCallData_dca) public nonReentrant returns (bytes32 dcaId) {
    if (_dca.isGelatoWatching) revert DcaMustNotBeStartedAtCreation(); // Setup the DCA
    dcaId = _setupDca(_dca); if (ownerOf[dcaId] != address(0)) revert DcaAlreadyExist(dcaId);
    ownerOf[dcaId] = msg.sender; dcas[dcaId] = _dca; dcas[dcaId].lastExecuted = uint120(block.timestamp -
    _dca.interval); ... } Figure 26.1: The createDca function in NestedDca.sol
```

The task ID should be assigned only by Gelato when startDca is called. It can be assigned only once and is meant to be unique to one created DCA task. function startDca(bytes32 dcaId, address feeToken) public override onlyDcaOwner(dcaId) returns (bytes32 taskId) { DcaStorage dca = dcas[dcaId]; if (dca.isGelatoWatching) revert DcaAlreadyStarted(dcaId); hashey 62 Nested Finance Security Assessment PUBLIC

```
... taskId = IOps(ops).createTask( address(this), abi.encodeCall(this.performDca, (dcaId)), moduleData,
feeToken ); dca.isGelatoWatching = true; dca.taskId = taskId; emit DcaStarted(dcaId, taskId, feeToken); }
Figure 26.2: The startDca function in NestedDca.sol
```

Because the code does not ensure that task IDs have been set only by Gelato, an attacker could claim to be the creator of any existing task or of a task that does not exist. This could break the system's integration with third-party components. Exploit Scenario Alice has a bot that watches the task ID of every DCA. Bob creates a DCA and a task ID of 0x41414141. Eve creates a new DCA and sets the task ID to 0x41414141. Alice's bot was not designed to handle multiple DCAs with the same task ID, so it crashes. Recommendations Short term, implement our recommendations for an alternative schema for storing DCAs in the NestedDca contract, proposed in appendix C.

Long term, document the expected validation that should occur when a DCA is created. Create a diagram highlighting the lifecycle of a DCA and the underlying invariants. Create unit tests for each state transition, and consider using Echidna to test multiple-transaction invariants. hashey 63 Nested Finance Security Assessment PUBLIC

27. restartDca lacks data validation Severity: Medium Difficulty: Medium Type: Data Validation Finding ID: TOB-NESTED-27 Target: NestedDca.sol Description The restartDca function has similar functionality to the createDca function; however, it lacks important safety checks and could be abused. restartDca behaves similarly to createDca. It sets up a DCA and stores its parameters and owner.

```
function restartDca(bytes32 dcaId, DcaCallData_dca) public override
onlyDcaOwner(dcaId) returns (bytes32 newDcaId) { // Setup the DCA
    newDcaId = _setupDca(_dca);
    if (dcaId == newDcaId) revert NothingToRestart(dcaId); // Erase the previous DCA
    delete dcas[dcaId];
    delete ownerOf[dcaId]; // Store the new DCA
    dcas[newDcaId] = _dca;
    dcas[newDcaId].lastExecuted = uint120(block.timestamp - _dca.interval);
    ownerOf[newDcaId] = msg.sender;
    emit DcaRestarted(dcaId, newDcaId, msg.sender, _dca.swapsParams); } Figure 27.1: The restartDca
function in NestedDca.sol The following are the steps in which restartDca differs from createDca :
```

1. It does not check whether the .isGelatoWatching parameter is already set.
2. It does not check whether the new DCA already exists and is running.
3. It does not set the maximum token allowance for the relevant tokens.
4. It does not check whether dca.swapsParams[i].sqrtPriceLimitX96 is 0 . hashey 64 Nested Finance Security Assessment PUBLIC

The missing check in step 1 allows users to set the .isGelatoWatching and .taskId system parameters. Any user could stop any DCA task by resetting an existing DCA, spoofing the task ID, and setting isGelatoWatching to true .

The missing check in step 2 allows users to overwrite an existing DCA. Without this check, users can disrupt an existing DCA by setting `isGelatoWatching` to false again or `taskId` to the empty bytes32 value. By resetting the `taskId`, the original task will not be associated with a DCA anymore, and it will not be possible to cancel it. Furthermore, the missing approvals in step 3 mean that the `performDca` function will fail if the tokens for the new DCA differ and if the `NestedDca` contract has not previously given tokens pending approval for these tokens. **Exploit Scenario** Alice sets up a task to dollar-cost-average her 100 million Token A back to USD over the next month. Eve stops Alice's DCA task. After one month, Token A's value suddenly declines to nearly zero. Without her knowledge, Alice's token has lost all of its value, since the DCA was not performed after being stopped by Eve without Alice's approval. **Recommendations** Short term, implement the four missing steps described in this finding in `restartDca`. Long term, document the expected validation that should occur when a DCA is created. Create a diagram highlighting the lifecycle of a DCA and the underlying invariants. Create unit tests for each state transition, and consider using Echidna to test multiple-transaction invariants. `hashey 65NestedFinanceSecurityAssessment PUBLIC`

28. Reentrancy vulnerabilities in `NestedDca` could allow Gelato operators to drain leftover tokens **Severity: Low Difficulty: High Type: Data Validation Finding ID: TOB-NESTED-28 Target: NestedDca.sol Description** The `performDca` and `updateDcaAmount` functions contain a reentrancy vulnerability that could allow malicious Gelato operators to drain leftover tokens. The `performDca` function lets Gelato operators perform DCA tasks. The `safeTransferFrom` function is called before `_performDcaSwap` to transfer the number of tokens specified in the `swapParams.amountIn` variable: `IERC20(swapParams.tokenIn).safeTransferFrom(ownerOf[dcaId], address(this), swapParams.amountIn); // Perform the DCA programmed swap amountsOut[i] = _performDcaSwap(dcaId, i);` **Figure 28.1:** `NestedDca.sol#L312-L317` The `_performDcaSwap` function calls the `UniswapRouter` and swaps the number of tokens specified in `swapParams.amountIn`: `function _performDcaSwap(bytes32 dcaId, uint256 swapParamsIndex) private returns(uint256 amountOut) { ISwapRouter.ExactInputSingleParams memory swapParams = dcas[dcaId].swapsParams[swapParamsIndex]; amountOut = uniswapRouter.exactInputSingle(swapParams);` **Figure 28.2:** `NestedDca.sol#L401-L407` The DCA's owner can change the value of `swapParams.amountIn` at any time by calling the `updateDcaAmount` function: `function updateDcaAmount(bytes32 dcaId, uint256 _amount) public override hashey 66NestedFinanceSecurityAssessment PUBLIC`

`onlyDcaOwner(dcaId) { if(_amount == 0) revert InvalidDcaAmount(_amount); for(uint256 i; i < dcas[dcaId].swapsParams.length; ){ ISwapRouter.ExactInputSingleParams storage swapParams = dcas[dcaId].swapsParams[i]; swapParams.amountIn = (_amount * dcas[dcaId].tokensOutAllocation[i]) / 10000;` **Figure 28.3:** `NestedDca.sol#L231-L236` However, there is no reentrancy protection in either `performDca` or `updateDcaAmount`. If a token being transferred has a callback mechanism, a malicious Gelato operator could do the following: • Make `performDca` transfer a small number of tokens • Reenter into `updateDcaAmount` to change `swapParams.amountIn` to be a larger number • Trigger the `Uniswap` swap with more tokens than were sent through the DCA task **As a result**, a malicious Gelato operator could steal tokens that are left in the contract. **Exploit Scenario** Eve is a malicious Gelato operator. She notices that \$10,000 worth of Token A are stuck in the `NestedDca` contract. Token A has a callback mechanism. Eve exploits the reentrancy vulnerability in `performDca` to steal the tokens. **Recommendations** Short term, add the `nonReentrant` modifier to `performDca` and `updateDcaAmount`. Long term, carefully evaluate every function that does not follow the check-effect-interaction pattern. Document any known reentrancy risks. `hashey 67NestedFinanceSecurityAssessment PUBLIC`

29. Unclear usage of `Dca.tokensOutAllocation` **Severity: Informational Difficulty: Low Type: Undefined Behavior Finding ID: TOB-NESTED-29 Target: NestedDca.sol Description** The documentation of the `tokensOutAllocation` parameter of the `Dca` struct is unclear and does not match the current implementation. The `tokensOutAllocation` parameter, a `uint16` array, is defined in the `Dca` struct. `struct Dca { uint120 lastExecuted; uint112 interval; uint16 swapSlippage; bool isGelatoWatching; uint16[] tokensOutAllocation; bytes32 taskId; ISwapRouter.ExactInputSingleParams[] swapsParams; }` **Figure 29.1:** The `Dca` struct definition in `INestedDca.sol` The documentation indicates that the values in `tokensOutAllocation` are the allocation percentage for each token. `/** tokensOutAllocation **: DCA's allocation percentage for each token` **Figure 29.2:** The description of the DCA data structure in `README.md` It is, however, not a required parameter because the number of tokens to be transferred are stored in the `tokensOutAllocation` parameter. The `tokensOutAllocation` parameter is used only in the `updateDcaAmount` function, in which the `swapParams.amountIn` parameter is updated. `/// @notice Update a DCA input tokens amount to swap one each period. /// @param dcaId The DCA bytes32 ID. /// @param amount The new DCA amount to swap.`

```
function updateDcaAmount(bytes32 dcaId, uint256 _amount) public override onlyDcaOwner(dcaId) {
    if(_amount == 0) revert InvalidDcaAmount(_amount);
    hasheye 68NestedFinanceSecurityAssessment PUBLIC
```

```
    for(uint256 i; i < dcas[dcaId].swapsParams.length; i++){
        ISwapRouter.ExactInputSingleParams storage swapParams = dcas[dcaId].swapsParams[i];
        swapParams.amountIn = (_amount * dcas[dcaId].tokensOutAllocation[i]) / 10000;
        unchecked{ ++i; }
    }
}
Figure 29.3: The updateDcaAmount function in WalletFactory.sol. The amounts in tokensOutAllocation are computed fractions of the amount given as a parameter. The exact percentages for each token are stored in tokensOutAllocation, which is not changeable without recreating the DCA. This introduces more complexity and gas usage and limits the number of possible values that the new amounts can take.
```

Furthermore, it assumes that all of the input tokens in swapParams are the same. Recommendations: Short term, remove tokensOutAllocation and allow the token amounts for DCA tasks to be given in absolute quantities.

Long term, carefully consider every feature added to a smart contract. Make sure the feature's goals are clear and necessary. hasheye 69NestedFinanceSecurityAssessment PUBLIC

#### 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
--------------------	--

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
-------------------	------------------------

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.
------	---

70NestedFinanceSecurityAssessment	PUBLIC
-----------------------------------	--------

#### 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 control 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 code-based documentation
---------------	---

Transaction Ordering Risks	The system's resistance to front-running attacks
----------------------------	--

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
--------------------------	--

Rating Criteria	Rating Description
-----------------	--------------------

Strong	No issues were found, and the system exceeds industry standards.
--------	--

SatisfactoryMinorissues that may affect systems safety were found. hashey 72NestedFinanceSecurityAssessment PUBLIC

WeakMany issues that affect systems safety were found.  
MissingAre required component is missing, significantly affecting systems safety.  
NotApplicableThe category is not applicable to this review.  
NotConsideredThe category was not considered in this review. Further Investigation Required  
Further investigation is required to reach a meaningful conclusion. hashey 73NestedFinanceSecurityAssessment PUBLIC

### C. Recommendations for NestedDca's Data Schema

Several issues described in this report stem from an incorrect schema used to store DCAs in NestedDca . This appendix proposes an alternative strategy that will reduce the code's complexity and the underlying risks. Current Schema DCAs are represented by the Dca structure: struct Dca { uint120 lastExecuted; uint112 interval; uint16 swapSlippage; bool isGelatoWatching; uint16[] tokensOutAllocation; bytes32 taskId; ISwapRouter.ExactInputSingleParams[] swapsParams; } Figure C.1: INestedDca.sol#L24-L32 These fields are set by either the user or the NestedDca contract: • The interval , swapSlippage , tokensOutAllocation , and swapsParams fields are meant to be controlled by the users. • The lastExecuted , isGelatoWatching , and taskId fields are meant to be set by NestedDca , following the DCA lifecycle. Every created DCA is stored in the dcas mapping: ///@notice Map of the DCA sid with all their information. ///If you want to get a DCA details, use getDca function. the solidity ///compiler does not support returning struct with arrays using ///mappings. mapping(bytes32 => Dca) private dcas; Figure C.2: NestedDca.sol#L78-L82 The key of the mapping is generated by hashing all of the parameters of the Dca structure: dcaId = keccak256(abi.encode(msg.sender, \_dca)); Figure C.3: NestedDca.sol#L420 This schema introduces several risks, including the following: hashey 74NestedFinanceSecurityAssessment PUBLIC

- Parameters that are meant to be set by the system can be directly set by users. This requires the contract to perform additional checks to ensure that they have not been set by users, which could result in issues (see findings TOB-NESTED-23 and TOB-NESTED-26). • DCAs can be recreated after they are restarted or their ownership is dropped (see findings TOB-NESTED-23 and TOB-NESTED-27). • Users are not able to create two DCAs with the same parameters. Proposed Schema Instead of hashing the structure, use a global counter. The createDca function would take only the user-controlled parameters and directly set the system parameters to their default values: createDCA(interval, swapSlippage, tokensOutAllocation, swapsParams) { dcaId = dcaCounter++; //note: it could be a user-level counter if needed .. //DCA validation dca[dcaId] = Dca( 0, interval, swapSlippage, false, tokensOutAllocation, 0x0, swapsParams ) Figure C.4: Pseudocode for createDca This schema would make it easier to validate the user-controlled input and would prevent system-controlled variables from being affected by users. hashey 75NestedFinanceSecurityAssessment PUBLIC

### D. Code Quality Findings This appendix lists findings that are not associated with specific vulnerabilities.

HyVM • Update the signature of the CONTINUE macro. The signature states that the macro returns one element, but it returns no elements. NestedWallet • Be explicit when referencing multiple overloaded functions in the NestedWallet contract. To avoid confusion, the code in figure D.1 could be changed to the code in figure D.2: address sender = isERC2771Activated()?\_msgSender():Context.\_msgSender(); Figure D.1: NestedWallet.sol#L80 address sender = isERC2771Activated()?ERC2771Context.\_msgSender():Context.\_msgSender(); Figure D.2: Alternative to the code in figure D.1 • Reconsider the use of ETH in NestedWallet . This contract was designed not to hold any ETH. However, users are allowed to unwrap ETH, but are unable to easily transfer it out. This is because the call helper functions require the recipient to contain code when calls with values are made. Vesting • Document the use of the delegate function. This function allows the beneficiary of vested tokens to delegate the vesting voting power, but it is not documented outside of the Solidity code. • Rename the onlyTeam modifier. The modifier's current name could mislead users into thinking that the modifier performs an access control, while it does not. Consider alternative names, such as allFeaturesEnabled or onlyIfTeamBeneficiaryIsEnabled . • Correct the NatSpec comments surrounding the Vesting.setLeaver function in src/Vesting.sol (lines 174-178). This comment refers to a specific period of time to increase the vesting schedule ("15 months") when calling setLeaver , but this period is no longer relevant as the duration is variable when the contract is initialized. hashey 76NestedFinanceSecurityAssessment PUBLIC

DCA • Reduce the complexity of the setMaxAllowance function or justify its purpose. This function has an unclear goal and purpose. Testing and Deployment • Add validation for the input of the HuffHelper.getHuffByteCode function.

This function will not work correctly if the file path used contains spaces or special characters, resulting either in a revert or even an incorrect deployment of the bytecode compiled by the Huff compiler. hashey 77NestedFinanceSecurityAssessment PUBLIC

E. Fix Review Results When undertaking a fix review, hashey reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From April 24 to April 26, 2023, hashey reviewed the fixes and mitigations implemented by the Nested Finance team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 25 issues described in this report, Nested Finance has resolved 12 issues, has partially resolved two issues, and has not resolved the remaining two issues. In addition, none of the nine DCA issues were resolved, as Nested Finance decided to rewrite this codebase. For additional information on the fix statuses of select issues, please see the Detailed Fix Review Results below. HyVM ID Title Severity Status

1 Unprotected call code allows anyone to destroy the HyVM master contact High Resolved (PR #39)

2 Lack of contract existence check on low-level calls High Partially resolved (PR #102)

7 Risk of overflow in FIX\_MEMOFFSET that could allow attacker to overwrite protected memory High Resolved (PR #40) 8 Users can directly call the HyVM implementation Informational Resolved (PR #36) 9 State-

changing operations permitted in HyVM Informational Unresolved 10 Lack of completed documentation on the known HyVM/EVM divergences Informational Resolved (PR #38) 11 Outdated HyVM used by nested-core-tetris Informational Resolved (PR #107) hashey 78NestedFinanceSecurityAssessment PUBLIC

12 Risks associated with using the Huff programming language Informational Partially resolved (PR

#33, PR#108, PR#104) TetrisCore ID Title Severity Status 13 Timelock Controller Emergency could be used to allow anyone to execute proposals Low Resolved (PR #101) 14 Risks associated with the use and design of the

Huff proxy Informational Unresolved 15 Use of tx.origin for access controls Medium Resolved (PR #100)

16 Ability to create wallets before WalletFactory is initialized Low Resolved Token ID Title Severity Status

17 Risk of portfolio voting power theft High Resolved (PR #10) 18 acceptDeal adds unreleased token to new vesting schedules Low Resolved (PR #13) 19 Vesting contract owner is not updated when VestingFactory owner changes

Informational Resolved 20 Unbounded loops could cause denial of service for third-party integrations Low Resolved (PR #11) DCA Issues related to the DCA codebase were not addressed. Nested Finance stated the following: hashey 79NestedFinanceSecurityAssessment PUBLIC

nested-dca has evolved a lot since its creation; it was audited when it was not stable enough. Significant changes just before the audit led to many errors and poor test coverage. Even after this audit, we cannot consider it to be stable and secure.

We believe we have chosen the wrong architecture/design, it will not go in production and will be archived. We will not apply the recommendations for all the issues regarding the DCA for this reason. We have started new developments for a new version that is outside the scope of the review.

PR <https://github.com/NestedFi/nested-core-tetris/pull/105>

Given the number of issues found within the DCA codebase, hashey agrees with this approach.

Detailed Fix Review Results Additional context for the fix statuses of select issues is provided in this section.

TOB-NESTED-2: Lack of contract existence check on low-level calls

Partially resolved in PR#102. The issue was fixed in Tetris but not in the proxy. Nested Finance stated the following:

Regarding the various points mentioned in the report, we will not address the problem in the same manner. In the context of Proxy.huff, we wish to retain the unverified call, as there is already a verification of the ImplementationResolver address from the WalletFactory, which passes this address to our Proxy. For function DelegateCallUnverified, we removed it and used function DelegateCall for calls to the HyVM, despite the optimization it provided (see PR).

Lastly, the HyVM will not incorporate verification directly at the opcode level, as this is achievable by adding the logic directly to the call to the HyVM. TOB-NESTED-9: State-

changing operations permitted in HyVM Unresolved. Nested Finance stated the following:

It is conceivable to prevent modifications to the state of a NestedWallet; at present, we do not require this functionality. If such a need arises, we have the option of employing a storage contract (accessed via a call). Nonetheless, this feature may pertain more to the NestedWallet rather than the HyVM, which we aim to keep as flexible as possible. We must carefully consider this matter. hashey 80NestedFinanceSecurityAssessment PUBLIC

TOB-NESTED-11: Outdated HyVM used by nested-core-tetris

Resolved in PR#107. While the immediate issue has been fixed, we recommend that

Nested Finance document a process to ensure that the dependencies stay properly up to date. TOB-NESTED-

12: Risks associated with using the Huff programming language

Partially resolved in PR#33, PR#108, and PR#104. Nested Finance stated the following:

We partially followed the recommendation by documenting the version used. It is not

possible to pin consistently the version in CI at this time. The creation of reference implementations in Solidity for every contract was done for proxy.huff but not for HyVM . TOB-NESTED-14: Risks associated with the use and design of the Huff proxy Unresolved. Nested Finance stated the following: We prefer to keep our architecture with proxy.huff without following the recommendations. TOB-NESTED-16: Ability to create wallets before WalletFactory is initialized Resolved. Nested Finance pointed out that this issue was a false positive due to the initialization of the nonReentrant modifier: Here, if we call WalletFactoryProxy::create() while the WalletFactory proxy ( Proxy.huff ) has not yet been initialized, it reverts with the Reentrancy() error. Indeed, the modifier ReentrancyGuardUpgradeable::nonReentrant() needs to be initialized in order to work properly, if that's not the case, it always reverts with the Reentrancy() error. And this ReentrancyGuardUpgradeable is initialized by the WalletFactoryProxy::initialize function in \_reentrancyGuard\_init() . This issue doesn't seem correct. We cannot create a wallet without having initialized the WalletFactory proxy. However, it does indeed revert with Reentrancy() , when it should revert with something like Locked() . While the reentrancy indeed prevents this issue from being triggered, we highly recommend refactoring the logic to not rely on this ad hoc protection. TOB-NESTED-19: Vesting contract owner is not updated when VestingFactory owner changes Resolved. Nested Finance stated that this behavior is intended: hashey 81NestedFinanceSecurityAssessment PUBLIC

No link between the Vesting owner and the VestingFactory owner, the behavior is intended. During the fix review, hashey asked if this behavior was documented, to which Nested Finance responded with the following: I clarify in the Vesting contract's natspec that the owner is the company, so if the VestingFactory passes its owner during the deployment of the Vesting to become the owner of the same vesting, we assume that the owners are always the company (for both contracts). If an ownership transfer needs to be done, it could be done atomically (if the owner is a smart contract like multisig). I can add documentation to specify that the change could be made contract by contract, it's not a problem. A de-synchronization of the ownership between the VestingFactory and the Vesting contracts is tolerated, since it will not really impact the beneficiary. hashey 82NestedFinanceSecurityAssessment PUBLIC