HashEye

// SMART CONTRACT SECURITY AUDIT

# Elixir Contracts

Security assessment by HashEye · prepared for Elixir Protocol

HASHEYE AUDITED

| | |
|---|---|
| PROJECT | Elixir Contracts |
| CLIENT | Elixir Protocol |
| CATEGORY | Blockchain |
| PUBLISHED | September 1, 2023 |
| REPORT ID | research-elixir-contracts-2023-09-01-11b84n |

This report was produced under HashEye's layered review process — **automated detection**, **pattern correlation**, and **senior manual verification** — with every finding signed off by a human reviewer. Full findings detail and on-chain attestation are available on the report page at hasheye.io/audits/research-elixir-contracts-2023-09-01-11b84n.

Vertex and Injective Contracts Security Assessment (Summary Report) September 28, 2023 Prepared for: Chris Gilbert Elixir Protocol Prepared by: Jaime Iglesias and Artur Cygan

About HashEye Founded in 2012 and headquartered in New York, HashEye provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high- end security research with a real -world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel. We maintain an exhaustive list of publications at https://github.com/hasheye-io/publications , with links to papers, presentations, public audit reports, and podcast appearances. In recent years, HashEye consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon. We specialize in software testing and code review 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. HashEye 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 hasheye on Twitter and explore our public repositorie s at https://github.com/hasheye-io . To engage us directly, visit our "Contact" pag e at https://www.hasheye.io/contact , or email us at info@hasheye.io . HashEye, Inc. 228 Park Ave S #80688 New York, NY 10003 https://www.hasheye.io info@hasheye.io HashEye 1 Elixir Protocol Security Assessment PUBLIC

Notices and Remarks Copyright and Distribution © 2023 by HashEye, Inc. All rights reserved. HashEye hereby asserts its right to be identified as the creator of this report in the United Kingdom. This report is considered by HashEye to be public information; it is licensed to Elixir Protocol under the terms of the project statement of work and has been made public at Elixir Protocol's request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of HashEye. The sole canonical source for HashEye publications is the HashEye 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 HashEye 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 codebase. HashEye 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. HashEye 2 Elixir Protocol Security Assessment PUBLIC

Table of Contents  HashEye 3 Elixir Protocol Security Assessment PUBLIC

Project Summary Contact Information The following project managers were associated with this project: Anne Marie Barry , Project Manager annemarie.barry@hasheye.io The following engineering directors were associated with this project: Josselin Feist , Engineering Director, Blockchain josselin.feist@hasheye.io The following consultants were associated with this project: Jaime Iglesias , Consultant Artur Cygan , Consultant jaime.iglesias@hasheye.io artur.cygan@hasheye.io Project Timeline The significant events and milestones of the project are listed below. Date Event September, 2023 Pre-project kickoff call September 13, 2023 Delivery of report draft September 13, 2023 Report readout meeting September 28, 2023 Delivery of summary report HashEye 4 Elixir Protocol Security Assessment PUBLIC

Executive Summary Engagement Overview Elixir Protocol engaged HashEye to review the security of the Vertex and Injective smart contracts. Vertex contracts are Solidity contracts deployed on Arbitrum that manage the Vertex DEX integration. Injective contracts are CosmWasm contracts that operate on

the Injective blockchain. A team of two consultants conducted the review from September 5 to September 11, 2023, for a total of two engineer-weeks of eff ort. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes. We covered all client code in scope and reviewed the relevant parts of the protocol integrations. We focused our time on assessing whether the contracts were behaving as expected, looking for common pitfalls (e.g., Solidity overflow) and adherence to DeFi best practices (e.g., that the query to the price oracle is performed correctly). Additionally, we reviewed the relevant parts of each of the protocol integrations, looking for possible edge cases or missing checks that could result in unexpected behavior. Both Vertex and Injective protocols are complex and have low-quality documentation. Given the short duration of the engagement and the general lack of technical documentation around the particular nuances of the integrations, there is a higher chance of edge cases that we have not been able to identify. Observations and Impact On the Vertex integration side, most findings concern possible improvements in the design (TOB-ELIX-09) and confusing behavior, such as particular nuances of the Vertex implementation. For example, because the Vertex sequencer is being used, withdrawals are done in two steps (withdraw and claim), which can lead to a degraded user experience in certain cases (TOB-ELIX-04). The higher-severity findings are related to panics resulting from underflow when using token decimals (TOB-ELIX-08) and logical bugs, such as the incorrect accounting of the router balance on Vertex when performing withdrawals (TOB-ELIX-03). These findings expose blind spots in both the unit and fuzz testing suites, and should be addressed by including tokens with arbitrary decimal amounts to help identify edge cases such as panics or rounding errors. Other needed improvements include adding new invariants such as “two pools cannot share the same router” or test cases in which the deposit and withdrawals on Vertex are performed during Elixir withdrawals to ensure the accounting is performed correctly. HashEye 5 Elixir Protocol Security Assessment PUBLIC

On the Injective integration side, most findings are logical bugs (e.g., using the wrong order type, as in TOB-ELIX-13) or violations of DeFi best practices (e.g., not checking how old the price returned by an oracle is, as in TOB-ELIX-19). Additionally, there is no documentation on this particular integration, and most assumptions about how it should work have been inferred from the documentation of the Vertex integrations and from conversations with the client. Our findings expose the lack of a systematic approach to testing and the usage of mocks that often do not refl ect the actual functionality. We believe that this indicates a lack of information in the ecosystem regarding testing tools and best practices. Finally, there is no documentation of the system assumptions and properties. Recommendations Improve the documentation around the particularities of each one of the protocol integrations (i.e., Vertex and Injective). Thoroughly document the assumptions regarding how these integrations work and use them to drive testing further. Improve the unit and fuzz tests for the Vertex integration, especially around the blind spots identified during the engagement (e.g., using tokens with arbitrary decimals and including additional test cases in which deposits and withdrawals are processed through Vertex). Additionally, improve the overall testing around the Injective integration. The audit detected multiple blind spots (e.g., using the wrong order types or lack of checks on market creation). Consider some of the tools suggested in the maturity table (e.g., injective test-tube) to ensure the underlying Cosmos-SDK is being used for testing instead of mocks. HashEye 6 Elixir Protocol Security Assessment PUBLIC

Codebase Maturity Evaluation HashEye uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Defi ciencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs. We evaluated Vertex and Injective contracts separately due to them being separate codebases with different levels of maturity. Vertex Contracts Category Summary Result Arithmetic The codebase relies on Solidity’s 0.8 native overflow protection, and the arithmetic operations perform rounding consistently. However, we have identified issues related to panics due to underflow in calculations involving token decimals which highlights blind spots in the unit and fuzz tests. Moderate Auditing While most functions emit events, certain functions emit events even if no actions have been performed. This could lead to confusion if an off-chain system listens to them. Additionally, the documentation contains a section regarding the usage of ChainAnalysis’s CIR as an incident response mechanism. However, it would be benefi cial to expand this documentation to contain more details regarding how CIR will be used and what is expected from the Elixir team. (See appendix B on how to build an incident response plan for some relevant questions to answer.) Moderate Authentication / Access Controls All functions that require access controls perform them adequately; however, documentation of the identities of different actors in the system and how the pools are managed should be expanded. Satisfactory HashEye 7 Elixir Protocol Security Assessment PUBLIC

Complexity Management While most functionality is isolated into functions and helpers and code duplication is minimal, certain design decisions have made certain parts of the system overly

complicated. Examples include the getVertexBalance function, which loops through the Vertex queue for each token in the withdrawal, and the usage of arrays to keep track of pool information, which led to code paths that could be completely avoided if different design decisions were used (e.g., users were forced to state amounts, even if zero, for each token in a pool when depositing). Moderate Cryptography and Key Management The contracts do not perform cryptographic operations or manage keys. Not Applicable Decentralization The available documentation states that the privileged actors in the system will be a 4/5 multi-sig wallet composed of active core members of the Elixir team and that there are plans to decentralize ownership by building a role-manager contract on top of the VertexManager to segregate roles and tasks for each type of operation. The documentation also mentions that a DAO structure may be used. However, this part of the system was not under the scope of the review, and we therefore cannot speak to its degree of decentralization. Further Investigation Required Documentation Documentation is sufficient to reason about the purposes of the system, and it contains step-by-step descriptions of each of the main user execution flows (deposit, withdraw, and claim), which is very informative. However, given that the bulk of the protocol complexity is in the Vertex integration itself, and given the overall lack of documentation of the Vertex protocol, it is of utmost importance to thoroughly document the assumptions around how the integration works. This integration documentation should document aspects of the system such as: • Which types of transactions exist in the Vertex queue and whether each type could affect the Moderate HashEye 8 Elixir Protocol Security Assessment PUBLIC

calculation of the Vertex balance inside Elixir • Detailed assumptions around the Vertex queue and whether transactions inside of it are guaranteed to be successfully executed, or whether it is possible that they silently fail and are removed from the queue • Situations in which a degraded user experience could occur—for example, when multiple users are withdrawing from the system but not all of their transactions are processed by the Vertex sequencer, they may be unable to claim temporarily if another user claims before them Finally, documentation around the type of market making the Elixir protocol will perform is also important. For example, if leverage trading will be used, certain Vertex transaction types will be relevant for the Elixir protocol. (Note that in this case, the Elixir team explicitly mentioned that there will be no leverage trading.) Front-Running Resistance We did not identify edge cases in this category that could lead to loss of funds; however, we did identify cases that could lead to a degraded user experience because of the Vertex sequencer. For example, when multiple users withdraw from Elixir, only one or a few of the withdrawals are processed by Vertex, requiring users to wait to withdraw funds or allowing users to front-run each others' claims to "withdraw faster". Additionally, because the protocol uses an upgradeable mechanism through UUPS, additional care has to be taken to ensure the initialization of the logic contract is not front-run (e.g., by performing the upgrade and initialization in a single transaction and ensuring the deployment scripts fail if the initialization is not performed correctly). Finally, we recommend that the client thoroughly review the functionality of the protocol with a focus on front-running (and other MEV "techniques") and to document any potential issues and mitigations. Moderate Low-Level No low-level manipulation is performed by the contracts. Not HashEye 9 Elixir Protocol Security Assessment PUBLIC

Manipulation Applicable Testing and Verification Testing is very thorough; coverage is high, and invariant testing is included. However, we identified some blind spots in the testing suite, such as the lack of usage of arbitrary decimals for tokens. We recommend giving the fuzzer the ability to generate tokens with arbitrary decimals, as this will help find panics or unexpected behavior and further strengthen confidence in the protocol. Additionally, we identified incorrect accounting of deposits in the Vertex queue when a withdrawal is processed, which exposes blind spots in the testing and the invariants. This finding should be reviewed and new unit tests cases and invariants should be implemented to ensure the correctness of the protocol accounting. Another invariant, " should be added to the protocol: "two pools cannot share the same router." Doing so will help prevent future code changes that could break the protocol accounting. Finally, all protocol invariants and assumptions should be thoroughly documented to help drive testing further. Occasionally reviewing these invariants and looking for new ones will help strengthen the protocol's testing. Moderate HashEye 10 Elixir Protocol Security Assessment PUBLIC

Injective Contracts Category Summary Result Arithmetic We did not identify any issues involving arithmetic operations. The project uses the overflow-checks Cargo configuration option for release that prevents silent integer overflows. Satisfactory Auditing The contracts do not have explicit events; instead, attributes are appended to the response. No documentation regarding incident response plans was provided. See appendix B for recommendations on how to create such a plan. Moderate Authentication / Access Controls All actions that require access controls perform them adequately; however, we found no functionality to transfer ownership. Additionally, no documentation is available regarding the identities of privileged actors in the system or how the pools are managed. Moderate Complexity Management The structure of the code follows the standard CosmWasm project. There is a non-trivial amount of code duplication from the cw20-base that makes

the project overly complex. Some of the more involved calculations tend to be confusing because they pack many variables into arrays. Additionally, we identified instances in which further checks should be performed to avoid unexpected behavior or a degraded user experience (e.g., not checking whether a marker is active or transferring zero tokens). Moderate Cryptography and Key Management The contracts do not perform cryptographic operations or manage keys. Not Applicable Decentralization No information whatsoever is available regarding the decentralization properties of this part of the system or Further Investigation HashEye 11 Elixir Protocol Security Assessment PUBLIC

how the privileged actors are. Required Documentation The injective integration has no documentation whatsoever, aside from some instructions on how to run the injective testnet and perform some actions through the CLI. No documentation is available to build the project, nor is there any documentation regarding the assumptions of the system or how the Injective integrations should work. The maturity of this category is further weakened by the fact that the Injective protocol documentation itself is also lacking, especially for the CosmWasm modules. Weak Front-Running Resistance Trading orders are submitted through the injective protocol and thus have the same front-running resistance properties. While we did not identify issues in this category during the engagement, we did identify a front-running issue in the add_fee function during the finalization of the report (i.e., before the final version of the report was handed to the client) that could lead to depositors paying more fees than they should while front-runners avoid them. We attempted to further investigate this finding but did not identify additional issues. However, additional investigation is required to ensure no additional edge cases are present. Finally, we recommend that the client thoroughly review the functionality of the protocol with a focus on front-running (and other MEV “techniques”) and document any potential issues and mitigations. Further Investigation Required Low-Level Manipulation The code does not use any low-level primitives such as unsafe code or inline assembly. Not Applicable Testing and Verification The code contains unit tests for spot-vault and perpetual-vault; however, we found a lack of a systematic approach for testing (e.g., using wrong order types or allowing one of the assets to be valued at zero for spot vaults). Weak HashEye 12 Elixir Protocol Security Assessment PUBLIC

The maturity of this category is further weakened by the lack of documentation regarding test tooling in the ecosystem. We attempted to identify tools that would improve testing (e.g., tools that do not rely on mocks and instead use the underlying Cosmos-SDK) and found some that we believe could improve this aspect of the system, including osmosis test-tube and injective test-tube . Note that we have not used these tools. Finally, we recommend documenting the system’s assumptions and properties, as this would help to further drive testing. HashEye 13 Elixir Protocol Security Assessment PUBLIC

Summary of Findings The table below summarizes the findings of the review, including type and severity details. ID Title Type Severity 1 Choosing the same router for pools with shared assets could lead to loss of funds Data Validation Informational 2 The getVertexBalance function should be improved Data Validation Undetermined 3 Incorrect pool balance accounting can lead to loss of funds Data Validation High 4 Claims can behave unexpectedly in certain scenarios because of the Vertex sequencer Data Validation Informational 5 The duplicate deposit and withdrawal flow is confusing and could lead to user mistakes Data Validation Informational 6 USDC price is hard-coded to $1 Data Validation Medium 7 Consider adding upper limits to fees Data Validation Informational 8 Insufficient checks for token decimals can lead to unexpected panics Data Validation High 9 Improvements to pool structure and usage Patching Informational 10 Insufficient checks in addPool function Data Validation Low 11 Code duplication Patching Informational HashEye 14 Elixir Protocol Security Assessment PUBLIC

12 Error handling with unwrap Error Reporting Undetermined 13 Perpetual vault is using atomic orders instead of market orders Data Validation Medium 14 Consider performing zero checks on amounts Data Validation Low 15 Consider implementing additional checks for markets Data Validation Informational 16 Consider adding an ownership transfer function Configuration Informational 17 Hardcap is applied to shares per user instead of applying it to the vault Data Validation Low 18 Vault allows unbalanced deposits Data Validation Informational 19 Insufficient checks on oracle price query Data Validation Medium 20 Consider adding upper limits to fees Data Validation Informational HashEye 15 Elixir Protocol Security Assessment 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 Undefi ned behavior triggered within the system HashEye 16 Elixir Protocol Security Assessment PUBLIC

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. Diffi culty 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. HashEye 17 Elixir Protocol Security Assessment 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 controls to handle identification and authorization and to ensure safe interactions with the system Complexity Management The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defi ned functions Cryptography and Key Management The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution Decentralization The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades Documentation The presence of comprehensive and readable codebase documentation Front-Running Resistance 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. Satisfactory Minor issues were found, but the system is compliant with best practices. Moderate Some issues that may affect system safety were found. HashEye 18 Elixir Protocol Security Assessment PUBLIC

Weak Many issues that affect system safety were found. Missing A required component is missing, significantly affecting system safety. Not Applicable The category is not applicable to this review. Not Considered The category was not considered in this review. Further Investigation Required Further investigation is required to reach a meaningful conclusion. HashEye 19 Elixir Protocol Security Assessment PUBLIC

C. Incident Response Recommendations This section provides recommendations on formulating an incident response plan. ● Identify the parties (either specifi c people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.). ● Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug. ○ Consider documenting a plan of action for handling failed remediations. ● Clearly describe the intended contract deployment process. ● Outline the circumstances under which Elixir will compensate users affected by an issue (if any). ○ Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw. ● Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on. ○ Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific system components. ● Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers) and how it will onboard them. ○ Effective remediation of certain issues may require collaboration with external parties. ● Define contract behavior that would be considered abnormal by off-chain monitoring solutions. HashEye 20 Elixir Protocol Security Assessment PUBLIC

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop “muscle memory.”

Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run. HashEye 21 Elixir Protocol Security Assessment PUBLIC

D. Fix Review Results When undertaking a fix review, HashEye 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. On September 26, 2023, HashEye reviewed the fixes and mitigations implemented by the Elixir team for the issues identified in this report. We reviewed each fix to determine its eff ectiveness in resolving the associated issue. In summary, of the 20 issues described in this report, Elixir has resolved 17 issues, has partially resolved 2 issues, and has not resolved the remaining one. For additional information, please see the Detailed Fix Review Results below. ID Title Status 1 Choosing the same router for pools with shared assets could lead to loss of funds Resolved 2 The getVertexBalance function should be improved Partially Resolved 3 Incorrect pool balance accounting can lead to loss of funds Resolved 4 Claims can behave unexpectedly in certain scenarios because of the Vertex sequencer Resolved 5 The duplicate deposit and withdrawal flow is confusing and could lead to user mistakes Resolved 6 USDC price is hard-coded to $1 Resolved 7 Consider adding upper limits to fees Resolved 8 Insufficient checks for token decimals can lead to unexpected panics Resolved HashEye 22 Elixir Protocol Security Assessment PUBLIC

9 Improvements to pool structure and usage Resolved 10 Insufficient checks in addPool function Resolved 11 Code duplication Resolved 12 Error handling with unwrap Resolved 13 Perpetual vault is using atomic orders instead of market orders Resolved 14 Consider performing zero checks on amounts Resolved 15 Consider implementing additional checks for markets Partially Resolved 16 Consider adding an ownership transfer function Resolved 17 Hardcap is applied to shares per user instead of applying it to the vault Resolved 18 Vault allows unbalanced deposits Resolved 19 Insufficient checks on oracle price query Resolved 20 Consider adding upper limits to fees Unresolved HashEye 23 Elixir Protocol Security Assessment PUBLIC

Detailed Fix Review Results TOB-ELIX-1: Choosing the same router for pools with shared assets could lead to loss of funds. Resolved in PR #17 . A new invariant was added to the fuzz tests that checks whether any of the pools share the same router. This will help identify future code changes that could make this possible. TOB-ELIX-2: The getVertexBalance function should be improved. Partially resolved in PR #17 . Changes were made to the function to improve its overall structure. However, we think the proposed fix is more complicated than it should (e.g., it relies on helper storage mappings and performs multiple loops). We recommend that the Elixir team consider simpler alternatives, such as the one suggested in the finding, to make the function simpler and more effi cient. Additionally, the fix review revealed another finding in the withdraw function related to duplicated tokens that could lead to users withdrawing more tokens than they should because of incorrect accounting. Note that this issue was present in the original code; however, the code changes introduced made the issue easier to exploit as, originally, it would have required the setup of a pool with duplicated tokens—something the fix for finding 10 should prevent. Additional checks should be included to prevent duplicate tokens from being used when calling the function; this could be achieved by, for example, requiring that that list of tokens is ordered, which can be easily done off-chain. Below is an example of how an ordered list can be used to prevent duplicates. This requires ordering the list off-chain but is a very simple pattern. function foo(address[] memory tokens) external { address lastToken = address(0); for (uint256 i = 0; i < tokens.length; i++) { require(token[i] > lastToken, “no duplicates allowed”); lastToken = token[i]; } } Finally, this finding further exposes blind spots in the testing and fuzzing of the codebase. We recommend that the Elixir team spend additional development time working on further improvements to tests and invariants and on simplifying the function. TOB-ELIX-3: Incorrect pool balance accounting can lead to loss of funds Resolved in PR #17 . The balance now correctly accounts for deposits in the Vertex queue. HashEye 24 Elixir Protocol Security Assessment PUBLIC

TOB-ELIX-4: Claims can behave unexpectedly in certain scenarios because of the Vertex sequencer. Resolved in PR #17 . Fees are now charged per user instead of being cumulative across withdrawals. This will help minimize the chances of degraded user experience during the claim process; however, note that certain scenarios are unavoidable because of the Vertex sequencer, which should be documented. TOB-ELIX-5: The duplicate deposit and withdrawal flow is confusing and could lead to user mistakes. Resolved in PR #17 . Two distinct deposit flows were created: one for the perp vaults and another for the spot vaults removing the potential confusion. Additionally, the documentation around the depositing of funds into the protocol has been improved. TOB-XYZ-6: USDC price is hard-coded to $1. Resolved in PR #17 . Instead of hard-coding the USDC price to 1$, Vertex’s clearing house contract is queried to get the price, which ensures that Vertex is being relied upon for pricing assets. Finally, additional comments and notes were included in the

documentation explaining how Elixir uses Vertex's oracle. TOB-ELIX-7: Consider adding upper limits to fees. Resolved in PR #17 . An upper limit of 100 USDC has been added for fees. This helps reduce the risk that a compromised or malicious manager owner steals funds from their users by setting a very high fee. TOB-ELIX-8: Insufficient checks for token decimals can lead to unexpected panics. Resolved in PR #17 . An upper bound limit of 18 was selected for token decimals (this is the same limit as Vertex). Additionally, certain parentheses in some calculations have been removed to prevent them from panicking in certain scenarios (e.g., when one of the tokens has a greater number of decimals than the other). Finally, the unit tests were improved; however, note that the fuzz tests should incorporate tokens with arbitrary decimals. TOB-ELIX-9: Improvements to pool structure and usage. Resolved in PR #17 . Several changes were made to the pool structure to incorporate mappings. As a result, new checks have been incorporated in the functions and certain code paths have ceased to exist, which has improved the readability of the code. TOB-ELIX-10: Insufficient checks in addPool function. Resolved in PR #17 . Additional checks were included in the function to prevent targeting already existing pools and including duplicated tokens. TOB-ELIX-11: Code duplication. HashEye 25 Elixir Protocol Security Assessment PUBLIC

Resolved in commit 8a8d866 . Several instances of duplicate code have been removed from the codebase by using the cw20-base library. TOB-ELIX-12: Error handling with unwrap. Resolved in commit 6addb9c ; expect is used and a reason for the panic is provided. TOB-ELIX-13: Perpetual vault is using atomic orders instead of market orders Resolved in commit 2e86e48 . Market orders are now being used instead of atomic orders. TOB-ELIX-14 - Consider performing zero checks on amounts. Resolved in commit d252e3f and 0243dca . Additional checks have been included in both the perpetual and spot vaults to both drive the decision making of the contract and to provide more explicit errors. TOB-ELIX-15: Consider implementing additional checks for markets. Partially resolved in commit 5260f52 . Additional checks have been included during market creation to ensure that the vault is being created for an active market. However, additional checks can still be performed. These include ensuring that the perpetual vault is actually targeting a perpetual market; that the spot vault is not targeting a derivative market; or checking the market status during deposits to inform users that the market is no longer active. TOB-ELIX-16: Consider adding an ownership transfer function. Resolved in commit 3bd0afb . cw-ownable was introduced to the contract, which handles ownership in a standard way and includes a two-step transfer process. TOB-ELIX-17: Hardcap is applied to shares per user instead of applying it to the vault. Resolved in commit d369dab . The hardcap is now correctly applied to the vault and not the user's balance. TOB-ELIX-18: Vault allows unbalanced deposits. Resolved in commit 5b605b4 . The deposit check && was replaced for || , which prevents unbalanced deposits. TOB-ELIX-19: Insufficient checks on oracle price query. Resolved in commit 73e8ee9 . Prices are now valid for one minute and the block timestamp is compared against the timestamp returned by Pyth to ensure that the price is not stale. TOB-ELIX-20: Consider adding upper limits to fees. Unresolved. HashEye 26 Elixir Protocol Security Assessment PUBLIC

E. Fix Review Status Categories The following table describes the statuses used to indicate whether an issue has been sufficiently addressed. Fix Status Status Description Undetermined The status of the issue was not determined during this engagement. Unresolved The issue persists and has not been resolved. Partially Resolved The issue persists but has been partially resolved. Resolved The issue has been sufficiently resolved. HashEye 27 Elixir Protocol Security Assessment PUBLIC