Cyberscope

Audit Report

Dynex

January 2025

Repository https://github.com/dynexcoin/Dynex

Commit 617034473bfdb043d5dad3e90ac8c96bf75bc11a

Audited by © cyberscope

Table of Contents

Table of Contents **1**
Risk Classification **3**
Review **4**
Audit Updates 4
Source Files 4
Overview **10**
Audit Scope 10
Architecture 10
Subsystems Reviewed 10
Security and Performance Considerations 11
Code Quality and Maintainability 11
Further Insights 11
Findings Breakdown **12**
Diagnostics **13**
CNC - Configuration Not Checked 14
Description 14
Recommendation 14
CAI - Constructor Assignment Inefficiency 15
Description 15
Recommendation 15
DCB - Duplicate Conditional Branches 16
Description 16
Recommendation 16
FAND - Function Argument Naming Discrepancy 17
Description 17
Recommendation 17
FPO - Function Parameter Optimization 18
Description 18
Recommendation 18
ISAC - Implicit Single Argument Constructors 19
Description 19
Recommendation 19
LS - Local Shadowing 20
Description 20
Recommendation 20
RCC - Redundant Condition Check 21
Description 21
Recommendation 21
SAE - STL Algorithm Efficiency 22

Description 22
Recommendation 22
UMV - Uninitialized Member Variable 23
Description 23
Recommendation 23
UEL - Unsigned Expression Logic 24
Description 24
Recommendation 24
Summary 25
Disclaimer 26
About Cyberscope 27

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/dynexcoin/Dynex
Commit	617034473bfdb043d5dad3e90ac8c96bf75bc11a

Audit Updates

Initial Audit	17 Jan 2024
Corrected Phase 2	22 Jan 2025

Source Files

Filename	SHA256
DynexCNCore/IntrusiveLinkedList.h	c8c819a53d74ffd5ee4c199eb0944555292e44680e10c3bc19fbb1a687e74b6f
DynexCNCore/ICoreObserver.h	b73f297e55b5df7eba697f483ea4d124b40de214c082e5c0a588d3eed1b34800
DynexCNCore/CoreConfig.cpp	6de34b238e9a82dd4cc550497795f7d3a9f2677a54953786f1137d0e3f91da90
DynexCNCore/SwappedMap.h	2ed41e7c0de9f86a102c7bbaeae6d85822a043a1cf7bc3ead4acb0f37041204c
DynexCNCore/Difficulty.cpp	689919a264aa37218daeab9f2d89fa0ffaa9e8523738bc1e20e329933e818bf9
DynexCNCore/Core.cpp	dd56435a26dedede4eb72d9a0fc4bf9f9f64a7d246dba9de1fd6814dea85f715

DynexCNCore/TransactionPrefixImpl.cpp	b5750ac241776d0a69d57ceffe6db8869bb9f65b6a3fa4ea658f4b2c70040f97
DynexCNCore/DynexCNFormatUtils.h	d9fae35aae56d2eb419ee89118d86fa74135b0db8014d936d1d4bfe3e46c2c48
DynexCNCore/OnceInInterval.h	d50abde79a375011c76da02e76e6c2511877dbcbd4ada9a67e63e70b3aa0d405
DynexCNCore/Difficulty.h	98cec1d6141aef5d8b5f0d6e0bbeca845480b0088852f8be3d5d659a2543c8e6
DynexCNCore/TransactionUtils.h	6f6fb53dbc53eda2edd798d354f2d0a9e58d43d4c97a779159b4e2d917fd3b06
DynexCNCore/Auth.h	00011a41d2f329f709de75643297522a26e6434b125c170c0e4c93abaf296de0
DynexCNCore/Account.h	faffe31ee4958a8488291c2ac00801c7a91b818c4bddd89fda02318fae99f6f5
DynexCNCore/Checkpoints.h	aab77dff23ae4cce1f0dbd600b8abc44060c039cd31a37eeed9aae76b36c28ba
DynexCNCore/ICore.h	b6bfd6811befba3c561fa21622c873b75e10e11f5b3749f66fe402d39670a374
DynexCNCore/TransactionPool.cpp	87b6c3c2d393c942555cc5c4eb4f24a6572d6da8623888d76504dc47c50ac1f1
DynexCNCore/SwappedVector.h	030fd06b37ebd7823578f31618dd7fd0e7e846dc7d7c1d7f3fef279f315ace9e
DynexCNCore/DynexCNTools.cpp	44f12887b1f562fd15c44ca18f37dedd2c553711c86023d033f372a16fcd5151

DynexCNCore/DynexCNBasicImpl.h	a0934d65ba2f995ebf6b27624ff282166c11962d047605b740bd6978660cd9f8
DynexCNCore/Checkpoints.cpp	ea7b14d1d8a93e26bebc8db442d79b2900a3a6a3694b4181cc5c4dc70adce2b5
DynexCNCore/UpgradeDetector.h	f889f7062645a53b1610dba2c0b2c96aa299ec712e4fa3e5d2fdb5d47d35660e
DynexCNCore/Currency.h	cfbf6911063516e88ccf37a3315c8a7c439cfc4e0d9c4f58ef2844d1e4561f59
DynexCNCore/DynexCNSerialization.h	8986096693942e786ee1ca7a1390e5d745c98a7ed9f340f1fae4038055c3316d
DynexCNCore/IBlockchainStorageObserver.h	740203e8d4fe68b939c3ab4ac702bc1f1ba938ab047fedebc9a9761094fe4f54
DynexCNCore/ITxPoolObserver.h	290928600bf4ca16e13396757d7eee0f369892fec14f327d4e66875e4ca60742
DynexCNCore/Currency.cpp	6a161efca68d4e5d4e66b25b21d53b661482f835ebb556cf0d8d831fdda28f27
DynexCNCore/Transaction.cpp	f1e4bb80ade2b26d2a741bd2928f55a3da0644ae75245b235e2df485ac7c4a7b
DynexCNCore/TransactionApiExtra.h	6afa93862373743551951887b42a7fa10a2ad7799ba6049b69fc11abc6faf9b6
DynexCNCore/BlockIndex.h	d30079702a37c0f8f9c1bdcfb5bb33f1e047171175af83f7997ab2576fcd4868
DynexCNCore/BlockchainMessages.cpp	12beb38b0fdfebee81e1f376ff5511c5b06d2c906aef22b58185ffdd0b58d242

DynexCNCore/DynexCNTools.h	5d025c3bba3512906481e3bff95258ad62c65dcae7eeb88366c95d009212640c
DynexCNCore/Blockchain.cpp	be3c5f849f51ccd5543c589a1a66c817a5da8b84f8e487458b7a20c45b16c302
DynexCNCore/TransactionExtra.h	c9a3509d80a11a38850cad13da5c7d01e3db5e90875ec2f02ccb63af481d9bd4
DynexCNCore/Auth.cpp	e669bc47bf5af2abb07e9855a115b692c9f8cc872803593bd0786758306c70eb
DynexCNCore/BlockchainIndices.h	2189f8cb1bf27be3eedf94c4d8991d25464157b39bd73f5b45c698af3743d866
DynexCNCore/CoreConfig.h	700baadcf396e3b8fec9f724d90348ae19020d30508cd9f2bdc1ce342e67d039
DynexCNCore/DynexCNSerialization.cpp	ff9fdfed05069122530840bacd017ee0ce5cec0500f6da20c83ff058d1038886
DynexCNCore/BlockIndex.cpp	f42ef1752d2f41873c1e08f3a8df701641ef87dbd75404d47a6829bace0ce09f
DynexCNCore/VerificationContext.h	6e1fe4c2bac6386f2c246593531fb38a71942ea00e65ff717975e634e2050087
DynexCNCore/ITimeProvider.h	a3ea366c0afefa957fc3e25a996712656b5fa957a1299ded4caf52cf28d4435e
DynexCNCore/ITransactionValidator.h	27182e50485db28a38367dda94c403b42929d6e6ed646d302e1bf46486ef6444
DynexCNCore/BlockchainMessages.h	3aa7824471bea64d58ee621f42920d9211bc0c6fe99a596446b137794f195432

DynexCNCore/TransactionApi.h	93194f44d06adef5ef2ea51f0b52b37692d182010690f89def4e0c4364656989
DynexCNCore/SwappedVector.cpp	89bdc67fc2653462eec8d36e8a7b38904248ffbd0dbb0e766548c1425a5cf606
DynexCNCore/DynexCNBasic.cpp	8c340dd6adca8a842d1174ff66fafa2e35bd110b6269f2379e8aacddf16919da
DynexCNCore/TransactionUtils.cpp	e05ae23fccb35c780f69548c9181c1f5eb902774c3ebe7b3efe717c39dd6ad63
DynexCNCore/DynexCNBasicImpl.cpp	e442a3483d2b9c6e9d54e48522fc656c18e3dfa61425c0d5129051407c449fc4
DynexCNCore/Core.h	4295ef1ea092c0ad6d3834e8e7c7e6875d88f272987def729d79b9a33a0b5721
DynexCNCore/UpgradeDetector.cpp	0199ea866bc1ed45001d21c82efcfc5473b65d65314b2da7d7e070206dc92d1d
DynexCNCore/SwappedMap.cpp	0bac1feaff7f3c4cda37e8c530d2d170465a054c7330661ac5f04074e10372aa
DynexCNCore/Account.cpp	d5a353579106483c65ecd6cfe1b4aec88e93d7f1a25ca833b61938325aac4171
DynexCNCore/IBlock.h	89f586d11e6a8ebf53c869f3ca196786fe292f5944038dd443b8925efef4872c
DynexCNCore/ITimeProvider.cpp	b49b155e83d2a0649af922cd8521a97db39a66067a8cef5bd635d0565b3d44f6
DynexCNCore/BlockchainIndices.cpp	aa0b09c09168fe51896516bc61323049ff84b1635e31123a682fca4b4b0f4b5c

DynexCNCore/Blockchain.h	cfddac97efabc887e46b61ca7a7ff360a15eca0b1f93682dd68db578d884e40f
DynexCNCore/MessageQueue.h	786cd7bcbd91dc6a7757ef5518b60bbf1944e404108e4eae7823b321f3050

Overview

Dynex is a next-generation platform for neuromorphic quantum computing that leverages blockchain technology to create a decentralized and distributed computing network. This implementation involves a node, which acts as a critical component in the network by facilitating blockchain operations, transaction processing, and network communications. The codebase is primarily written in C++ and provides a modular structure to manage various aspects of node functionality.

At the current time of this report, 22 January 2025, the total circulating supply of DNX is 99.72 million, with a maximum supply of 110 million DNX tokens.

Audit Scope

This audit focuses on the C++ codebase of the Dynex node as extracted from the provided repository. The scope includes the evaluation of core blockchain operations, transaction management, network communication protocols, and cryptographic utilities. The objective is to ensure the security, performance, and maintainability of the node software while verifying adherence to modern C++ coding standards.

Architecture

The Dynex node is designed to support the platform's unique neuromorphic quantum computing requirements. The architecture employs specialized data structures and algorithms to optimize the processing and storage of blockchain data. Core components include blockchain validation, consensus mechanisms, and transaction pooling. The design ensures scalability and reliability to handle distributed computation workloads.

Subsystems Reviewed

The node's blockchain logic manages essential tasks like block validation, indexing, and chain state updates. The transaction pool subsystem handles the queuing and prioritization of transactions awaiting inclusion in blocks. Protocol handlers facilitate node-to-node communication, ensuring data consistency and synchronization across the network. Additionally, utilities for serialization, cryptographic functions, and configuration management enhance the node's robustness and flexibility.

Security and Performance Considerations

The security review emphasizes transaction integrity, cryptographic key management, and resistance to common blockchain vulnerabilities such as double-spending or Sybil attacks. Performance evaluations focus on transaction throughput, block processing efficiency, and effective resource utilization for distributed computations.

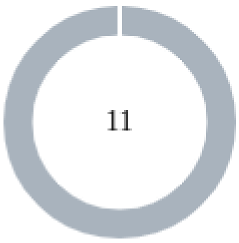
Code Quality and Maintainability

The codebase is assessed for modularity, readability, and adherence to modern C++ practices. Recommendations aim to optimize function implementations, eliminate redundancy, and improve maintainability. Clear documentation and logical structuring of components enhance the developer experience.

Further Insights

This implementation positions Dynex as a unique player in the blockchain domain, combining neuromorphic computing with decentralised technology. A deeper analysis comparing its innovations to other blockchain systems could provide valuable insights and help identify further areas for optimization or enhancement.

Findings Breakdown



- Critical 0
- Medium 0
- Minor / Informative 11

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	11	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	CNC	Configuration Not Checked	Unresolved
●	CAI	Constructor Assignment Inefficiency	Unresolved
●	DCB	Duplicate Conditional Branches	Unresolved
●	FAND	Function Argument Naming Discrepancy	Unresolved
●	FPO	Function Parameter Optimization	Unresolved
●	ISAC	Implicit Single Argument Constructors	Unresolved
●	LS	Local Shadowing	Unresolved
●	RCC	Redundant Condition Check	Unresolved
●	SAE	STL Algorithm Efficiency	Unresolved
●	UMV	Uninitialized Member Variable	Unresolved
●	UEL	Unsigned Expression Logic	Unresolved

CNC - Configuration Not Checked

Criticality	Minor / Informative
Location	DynexCNCore/Auth.cpp:84
Status	Unresolved

Description

The code includes a reference to the configuration parameter `CURLOPT_MAXAGE_CONN`, which is not explicitly defined or verified during the build process. This oversight can lead to scenarios where the parameter's value remains undefined, causing unexpected runtime behaviour or failing tests that depend on its proper configuration. Such issues can result in unpredictable execution and may complicate debugging efforts, especially in critical authentication workflows.

Recommendation

It is essential to define the configuration parameter explicitly during the build process, using `-D` or `-U` flags as needed. By ensuring the parameter is properly set or explicitly skipped, the code will exhibit consistent and predictable behaviour. Additionally, this practice helps identify misconfigurations early, enabling more robust testing and reducing the risk of runtime errors.

CAI - Constructor Assignment Inefficiency

Criticality	Minor / Informative
Location	DynexCNCore/CoreConfig.cpp:46 DynexCNCore/TransactionPool.h:77
Status	Unresolved

Description

In the file `DynexCNCore/CoreConfig.cpp:46`, the variable `configFolder` is assigned a value using the `Tools::getDefaultDataDirectory()` function within the constructor body. Similarly, in `DynexCNCore/TransactionPool.h:77`, the variable `m_lastWorkedTime` is set to `0` directly in the constructor body. Assigning values to member variables in the constructor body introduces unnecessary performance overhead as it involves an extra step of default initialization followed by reassignment. This approach also reduces code clarity, as the initial values of the variables are not immediately apparent when the constructor is invoked.

Recommendation

To address this inefficiency, the constructors in both files should be refactored to use initialization lists for member variables. This change will eliminate redundant default initializations, ensuring that member variables are directly initialized with their intended values. Using initialization lists not only improves performance but also enhances code readability and maintainability by clearly specifying the initial values of the variables.

DCB - Duplicate Conditional Branches

Criticality	Minor / Informative
Location	DynexCNCore/Currency.h:121 DynexCNCore/Currency.cpp:486
Status	Unresolved

Description

The code contains duplicate branches in conditional statements where the logic for both `if` and `else` branches is identical. This redundancy can confuse developers and obscure the intended logic. In the identified cases, both branches perform the same operations under different conditions, making the conditional structure unnecessary.

Recommendation

Refactor the conditional statements to eliminate duplicate branches. Either combine the conditions into a single block or rewrite the logic to reflect the intended behaviour more clearly. Removing such redundancies improves code clarity and reduces the potential for misunderstanding or unintended behaviour.

Recommendation

This separation ensures that each issue is addressed with the specificity and focus required for effective resolution.

FAND - Function Argument Naming Discrepancy

Criticality	Minor / Informative
Location	DynexCNCore/BlockchainIndices.cpp:53 DynexCNCore/Blockchain.cpp:797 DynexCNCore/Blockchain.cpp:1268
Status	Unresolved

Description

The code contains multiple instances where the names of function arguments differ between their declaration and definition. This discrepancy can lead to confusion and reduced readability, as developers may struggle to track and understand the intended purpose of the arguments. For example, argument names such as `h` and `blockHash`, or `start_height` and `start_top_height`, create inconsistencies that complicate code maintenance and debugging. This issue is prevalent across various functions and constructors in the codebase.

Recommendation

To improve clarity and maintainability, ensure that the argument names in function declarations match their corresponding definitions. Consistent naming helps developers understand the purpose of arguments at a glance, reduces cognitive load, and prevents potential errors when using or modifying the code. Standardising argument names across the codebase will enhance overall readability and align the implementation with best practices.

FPO - Function Parameter Optimization

Criticality	Minor / Informative
Location	DynexCNCore/Blockchain.h DynexCNCore/BlockchainIndices.cpp
Status	Unresolved

Description

The code contains multiple instances where function parameters are not optimized for efficiency. In `DynexCNCore/Blockchain.h` and `DynexCNCore/BlockchainIndices.cpp`, parameters such as `s` in various `serialize` functions are passed as non-const references. Since these parameters are not modified within the scope of the functions, they could be more efficiently declared as references to `const`. By not using `const` references, unnecessary data copying occurs, particularly when handling objects like serializers, which can lead to increased overhead and reduced performance.

This issue is prevalent across several serialization functions and could impact overall efficiency, especially in contexts where these functions are called frequently or handle large objects.

Recommendation

To enhance performance and improve code clarity, it is recommended to revise the parameter passing strategy for functions identified in the code. Parameters that are not modified should be passed as references to `const`, reducing unnecessary copying and ensuring consistency with modern C++ best practices. This approach is particularly beneficial for functions like `serialize`, where large objects or frequently used parameters can lead to significant performance improvements when passed as `const` references. Applying this change consistently across the codebase will improve maintainability and efficiency.

ISAC - Implicit Single Argument Constructors

Criticality	Minor / Informative
Location	DynexCNCore/Blockchain.h:412 DynexCNCore/BlockchainIndices.h:167 DynexCNCore/Checkpoints.h:48
Status	Unresolved

Description

The code contains several single-argument constructors that are not explicitly marked with the `explicit` keyword. This omission can lead to unintended implicit conversions, potentially introducing subtle and hard-to-diagnose bugs. When single-argument constructors are not marked as `explicit`, they allow the compiler to perform implicit type conversions, which may result in unexpected behaviour, particularly in scenarios where strict type safety is expected. Such constructors are present in various classes across multiple files, and their implicit nature may cause confusion or errors when these classes are used in type-sensitive operations.

Recommendation

To prevent unintended implicit conversions and ensure the safety and clarity of the code, all single-argument constructors should be explicitly marked with the `explicit` keyword. This change ensures that the compiler does not use these constructors for implicit conversions, making the code more robust and reducing the likelihood of bugs. Adopting this practice also improves code readability by making the intent of the constructors clearer to developers.

LS - Local Shadowing

Criticality	Minor / Informative
Location	DynexCNCore/SwappedMap.h:155 DynexCNCore/TransactionPool.h:153 DynexCNCore/UpgradeDetector.h:88 DynexCNCore/Currency.cpp:658
Status	Unresolved

Description

The code contains several instances where local variables or functions shadow others with the same name in outer scopes. This practice can lead to unintended behaviour and increased difficulty in understanding the code, as it is not always clear which variable or function is being referenced. Such shadowing is observed across different contexts, including variables within loops, temporary assignments, or even functions with overlapping names. These issues make the code harder to maintain and debug, as scope-related errors may occur when the intended variable or function is not accessed.

Recommendation

To improve code clarity and maintainability, it is recommended to rename shadowing variables and functions to ensure they do not conflict with names in outer scopes. Adopting unique and descriptive naming conventions will reduce the risk of ambiguity, making the code easier to understand and preventing errors caused by unintended scope references. By addressing these issues, the overall reliability and readability of the code will be enhanced.

RCC - Redundant Condition Check

Criticality	Minor / Informative
Location	DynexCNCore/Blockchain.cpp:1510
Status	Unresolved

Description

The code contains a condition that is always false, making it redundant. In this instance, the condition `!(!missed_tx_id.size())` is logically equivalent to `missed_tx_id.size() != 0`, but the earlier logic ensures that this condition will always evaluate to `false`. Such redundant checks do not add any functional value and can obscure the intent of the code, making it harder to understand and maintain. These types of conditions can mislead developers by suggesting potential scenarios that do not actually occur, increasing the cognitive load required to interpret the code.

Recommendation

It is recommended to review and refactor the identified condition to remove unnecessary complexity. Simplifying this condition or removing it entirely, if appropriate, will improve the code's readability and maintainability. Ensuring that only meaningful and logically sound conditions are used reduces the risk of confusion and helps maintain clear and concise logic throughout the codebase.

SAE - STL Algorithm Efficiency

Criticality	Minor / Informative
Location	DynexCNCore/BlockIndex.cpp:69 DynexCNCore/Blockchain.cpp:971, 1210, 1535, 2053
Status	Unresolved

Description

The code contains several instances across multiple files where raw loops are used, which could be replaced with more efficient and expressive Standard Template Library (STL) algorithms. For example, in `DynexCNCore/BlockIndex.cpp`, the loop used for checking conditions could leverage the `std::any_of` algorithm for better clarity and efficiency. Similarly, in `DynexCNCore/Blockchain.cpp`, there are opportunities to use `std::any_of`, `std::all_of`, or `std::none_of` for condition-based evaluations, while `std::accumulate` could replace raw loops for accumulating values such as rewards or cumulative sizes. The addition of elements to collections, as seen in the same file and in `DynexCNCore/Checkpoints.cpp`, could be rewritten using `std::transform` for transforming and adding elements in a single, optimised step.

Recommendation

To enhance efficiency, readability, and maintainability, the raw loops in the identified files should be refactored to utilise STL algorithms such as `std::any_of`, `std::accumulate`, and `std::transform`. These algorithms provide a more declarative approach to programming, allowing developers to clearly express intent without relying on verbose loop constructs. By updating the code to adopt these algorithms, it will become more concise, easier to maintain, and aligned with contemporary C++ coding standards. Leveraging STL algorithms also ensures that the code benefits from their optimised internal implementations, potentially improving performance.

UMV - Uninitialized Member Variable

Criticality	Minor / Informative
Location	DynexCNCore/SwappedMap.h:139 DynexCNCore/SwappedVector.h:213 DynexCNCore/SwappedVector.h:67
Status	Unresolved

Description

The code includes instances in multiple files where member variables are not properly initialised in constructors. These uninitialised variables can lead to undefined behaviour if accessed before explicit assignment. This issue introduces the risk of unpredictable program outcomes and may obscure potential bugs, particularly when the uninitialised variables are essential to the logic of the respective classes.

Recommendation

All member variables should be initialised in constructors to ensure a well-defined state for objects upon creation. This can be accomplished by using an initialisation list in the constructor or by setting default values within the class definition. Proper initialisation improves the reliability, maintainability, and predictability of the code while reducing the likelihood of bugs arising from uninitialised variables.

UEL - Unsigned Expression Logic

Criticality	Minor / Informative
Location	DynexCNCore/Currency.cpp:887, 914, 923
Status	Unresolved

Description

The code contains multiple instances where unsigned expressions are checked against being less than or equal to zero. Since unsigned data types can never be negative, these checks are logically redundant and may indicate a misunderstanding of the underlying data type. In the identified cases, the variable `val`, an unsigned expression, is being compared with `0`, making the condition always evaluate as `true` or redundant. These checks can obscure the logic, reduce code clarity, and may mislead developers regarding the purpose of the condition.

Recommendation

Remove redundant checks for unsigned expressions being less than or equal to zero, as they do not provide meaningful logic. If the intention is to verify that `val` is non-zero, the condition should be simplified to `val > 0` or just `val`. Ensuring logical correctness in such conditions improves code readability, eliminates unnecessary operations, and reduces the potential for confusion among developers.

Summary

The Dynex code implements a comprehensive node system designed for its neuromorphic quantum computing blockchain platform. This audit investigates security issues, business logic concerns, performance optimization opportunities, and adherence to best coding practices. The code review and auditing process have identified several key areas for improvement within the C++ codebase, focusing on the robustness, efficiency, and maintainability of the node implementation.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an “endorsement” or “disapproval” of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io