Cyberscope

Audit Report

Dynex

January 2025

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

Commit 617034473bfdb043d5dad3e90ac8c96bf75bc11a

Audited by © cyberscope

Table of Contents

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	20 Jan 2025

Source Files

Filename	SHA256
IFusionManager.h	27a6df6413a01321184729ece602eff733b2c6ad4cdf4f0e40e40f9d9041b5fe
LegacyKeysImporter.cpp	079bbaf239a363e8cbd810033ef5db1d4c16c9c8613acc04f32ff7016e6b9a68
LegacyKeysImporter.h	1147933c055017613c7456af9203aabb96f3cf8239ecbf8cc7be8c0d7de8d3c4
WalletAsyncContextCounter.cpp	e63a35b3deffa1d6653d605b6fcc7021595c8accdc8ba77fe83cac1a16b99c38
WalletAsyncContextCounter.h	13c3a262c7e28e9663499fb3b50ba057a0819da18ce9a0ec41820e1f4ad85fde
WalletErrors.cpp	8d0bf4cc7dd97fd2b8cc89ea140dc088f36555ad801d91d91b079bb12cce340a

WalletErrors.h	243ee5cc3b0152adea59bad3519dcaea85868f190b0191a3cc94fea663cc86c0
WalletGreen.cpp	4ab229f231451d2d2e205a42c3a977bc6d44fed9d9597eb30a17d7116733ae42
WalletGreen.h	87690960d91b94af66b960591f6e826a7ff0d8e1300969fb8c29504838235697
WalletIndices.h	dfdd2d9d873d8e9c109d66de7b750c7422bdd2580fdea1cbd5e53d3c6f2792c7
WalletRpcServerCommandsDefinitions.h	d6f41e6dac9c99fcc2f0e7fb8d80b08e572a33346a2f31d2f8e857fd58905074
WalletRpcServer.cpp	b72bfc5a09391dd9cd69fe934567b2d2feabce79f27e5837c0eb6aef5bd2f565
WalletRpcServerErrorCodes.h	1aef8eb45fa173df0da90dbfddac23c98e4cae7d8472cfd8b761adeef1c10910
WalletRpcServer.h	66abca7677b45c2fab27eff2528936a78f732fbb3e0a202977f74921a1861061
WalletSerializationV1.cpp	59aecf0294854eb8d5b34750a81cab73c9bea3d01f5de75d78e49373e0d8f322
WalletSerializationV1.h	86c1be1762f7430b50f7f371a85666ca2e448be1de21486e5264f8724a8bb688
WalletSerializationV2.cpp	11b98fde0e8f15ab3c85e556056474e3e2fd3efd45df5bd7cefefa320e53de3a
WalletSerializationV2.h	b2112c2789eae146329f45f6dc86884699a70b4c41d3d1b4625c34886b04a160

WalletUtils.cpp	8eb76e597bfd58e0ca5c003b579d70b2ead8b174bd9fe5f352088e94527c67d1
WalletUtils.h	10bd68a4ed4a55a0ec6edd5b8e332f2aedccaae11b53046b5f7439643b553413

Overview

Dynex, as described on its GitHub repository, is a next-generation platform for neuromorphic quantum computing based on a new flexible blockchain protocol. This technology involves a network of nodes contributing to a massive neuromorphic computing network, aiming for high-speed and efficient computations. The codebase is primarily in C++, supplemented by some C components.

Audit Scope

The audit scope of the C++ code audit focuses on the `Wallet` folder of the release `dynex_d30c774` of the repository. This includes a thorough examination of key areas crucial for functionality and security. The audit primarily concentrates on security vulnerabilities, potential optimizations for performance, code maintainability, and adherence to best coding practices. Additionally, the audit evaluates the robustness of error handling mechanisms, the efficiency of network communication protocols, and the proper use of dependencies and external libraries. Each of these aspects is assessed to ensure the overall reliability and security of the wallet functionality.

Architecture

The Dynex blockchain protocol, according to the repository, is designed for neuromorphic quantum computing, which suggests a unique architectural approach. This architecture might involve specialized data structures and algorithms optimized for neuromorphic processing. However, without direct access to the architectural documentation, the specifics of this architecture remain unclear from the GitHub overview.

Wallet Subsystem

The focus of this audit is the Wallet component of the Dynex project. In blockchain technology, a wallet typically handles key management, transaction creation, and sometimes node interactions. Understanding its implementation, security measures, and integration with the broader system is crucial.

Security and Performance Considerations

Given the Wallet's role in managing keys and transactions, security is paramount. This includes scrutinizing cryptographic implementations, key management, and transaction signing mechanisms. Additionally, performance aspects, especially in transaction processing and network communication, should be evaluated.

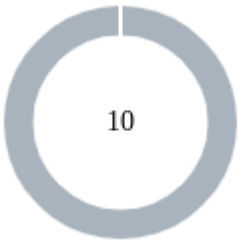
Code Quality and Maintainability

The Wallet code is reviewed for clarity, maintainability, and adherence to C++ best practices. This includes evaluating the use of modern C++ features, code modularity, and documentation.

Further Insight

A deeper dive into specific components or algorithms used in the Wallet, possibly comparing them with industry standards or similar implementations in other blockchain projects, would be beneficial.

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	10

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	10	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	AMV	Accessing Moved Variable	Unresolved
	CAI	Constructor Assignment Inefficiency	Unresolved
	FPO	Function Parameter Optimization	Unresolved
	ISAC	Implicit Single Argument Constructors	Unresolved
	LVS	Local Variable Shadowing	Unresolved
	RCC	Redundant Condition Check	Unresolved
	RVA	Redundant Variable Assignment	Unresolved
	SAE	STL Algorithm Efficiency	Unresolved
	UMV	Uninitialized Member Variable	Unresolved
	UAV	Unutilized Assigned Variable	Unresolved

AMV - Accessing Moved Variable

Criticality	Minor / Informative
Location	Wallet/WalletGreen.cpp:2908, 2910
Status	Unresolved

Description

In the provided code, the variable `wallets` is accessed after it has been moved, which can lead to undefined behaviour. This issue arises specifically in Wallet/WalletGreen.cpp when operations are performed on the moved variable, such as accessing the `spendSecretKey` and `spendPublicKey` properties. Once a variable is moved, its state becomes indeterminate, making further access potentially unsafe and unreliable. This finding applies only to scenarios where the variable `wallets` has already been moved before the access occurs.

Recommendation

To resolve this issue, the code should ensure that `wallets` is not accessed after it has been moved. If access is necessary after the move, creating a copy of the variable before moving it is advisable. Alternatively, restructuring the logic to avoid reliance on a moved variable will ensure program stability and prevent undefined behaviour. Ensuring that `wallets` is in a valid state before performing operations is essential for predictable execution.

CAI - Constructor Assignment Inefficiency

Criticality	Minor / Informative
Location	Wallet/WalletSerializationV1.cpp:93, 96, 101 Wallet/WalletSerializationV2.cpp:62, 65, 70, 95
Status	Unresolved

Description

Several variables in wallet-related files are assigned in the constructor body instead of being initialized using an initialization list. This practice is less efficient as it introduces extra assignments after default initialization. Examples include the `state`, `hash`, and `extra` variables in WalletSerializationV1.cpp and WalletSerializationV2.cpp. This approach can lead to unnecessary performance overhead and reduced code clarity.

Recommendation

Refactoring the constructors in these files to use initialization lists for member variables is recommended. This change eliminates redundant default initializations, improving performance and making the code cleaner and easier to maintain. Using initialization lists ensures that the initial values of member variables are explicitly specified, enhancing readability and efficiency.

FPO - Function Parameter Optimization

Criticality	Minor / Informative
Location	WalletSerializationV1.cpp:134,142,150,155,182,410 WalletSerializationV2.cpp:105,133,225,269,282,310,342,356,364,373,396
Status	Unresolved

Description

In the `WalletSerializationV1.cpp` and `WalletSerializationV2.cpp` files, there are multiple instances where function parameters are not optimized for efficiency. Parameters such as `address`, `serializer`, and others are frequently passed by value or as non-const references, leading to unnecessary data copying. This inefficiency is particularly pronounced in serialization functions and when handling large objects. Additionally, variables like `tx`, `walletsIndex`, and `index`, which are not modified within their respective scopes, could be more efficiently declared as references to const to avoid redundant copying.

Recommendation

To enhance performance and reduce overhead, it is recommended to revise the parameter passing strategy. Wherever possible, particularly in cases where parameters are not being modified, objects should be passed by const reference. This change will optimize the code by reducing unnecessary copying and improving efficiency, especially in the context of serialization functions and when dealing with large objects. The recommended modifications should be applied to `WalletSerializationV1.cpp`, `WalletSerializationV2.cpp`, and other similar instances throughout the codebase to ensure consistency and adherence to best coding practices.

ISAC - Implicit Single Argument Constructors

Criticality	Minor / Informative
Location	Wallet/WalletSerializationV1.cpp:92:3: Wallet/WalletSerializationV1.cpp:120:3
Status	Unresolved

Description

The code contains several single argument constructors that are not explicitly marked as `explicit`. This oversight can lead to unintentional implicit conversions, potentially causing subtle and hard-to-diagnose bugs. Such constructors, when not explicitly defined, allow the compiler to perform implicit conversions, which may introduce unexpected behaviour in the code, especially when handling type-sensitive operations.

Recommendation

To prevent unintended implicit conversions and enhance the safety and clarity of the code, it is recommended to mark all single argument constructors with the `explicit` keyword. This ensures that the compiler does not use these constructors in implicit conversions, thereby reducing the risk of subtle bugs and making the intent of the code more transparent.

LVS - Local Variable Shadowing

Criticality	Minor / Informative
Location	Wallet/WalletGreen.cpp:1689,1709,2256,2289
Status	Unresolved

Description

The code contains multiple instances of local variables shadowing other variables or functions with the same name but different scopes. This shadowing can cause confusion and increase the risk of unintended behaviour, as the intended variable may not be accessed as expected. In Wallet/WalletGreen.cpp, examples include the `transfer` variable, which is shadowed in different contexts, such as within loops or function scopes. These occurrences make the code harder to debug and maintain, as it becomes challenging to track which variable is being referenced in a given scope.

Recommendation

To enhance code clarity and maintainability, it is recommended to rename local variables to avoid shadowing outer variables or functions. Clear and unique naming conventions for variables ensure that the code behaves as intended and reduces the risk of errors caused by scope confusion.

RCC - Redundant Condition Check

Criticality	Minor / Informative
Location	Wallet/WalletGreen.cpp:1602,1610,1619,2506,2516
Status	Unresolved

Description

The code contains conditions that are always true or false, making them redundant. In Wallet/WalletGreen.cpp, the condition `mixIn != 0` is always false because `mixIn` is explicitly assigned the value `0` earlier in the code. Similarly, the condition `dust` is always false as it is assigned `false` before the check. These redundant checks can obscure the intent of the code, potentially leading to confusion and making it harder to identify logical issues.

Recommendation

It is recommended to review and refactor these conditions to reflect the intended logic accurately. Removing or rewriting redundant checks will improve code clarity and maintainability. Simplifying such conditions reduces cognitive load for developers and prevents misunderstandings caused by misleading or unnecessary logic.

RVA - Redundant Variable Assignment

Criticality	Minor / Informative
Location	WalletLegacy/WalletTransactionSender.cpp:576
Status	Unresolved

Description

The variable `selectOneUnmixable` in WalletLegacy/WalletTransactionSender.cpp is redundantly initialised and reassigned a value before its previous value is used. Specifically, `selectOneUnmixable` is initialised to `false` at line 576, but this value is overwritten shortly afterward. This redundant initialisation and reassignment do not contribute to the code's functionality and can lead to unnecessary confusion, making it harder to understand the code's intent.

Recommendation

It is recommended to review the logic involving `selectOneUnmixable` and remove the redundant initialisation and reassignment. The variable should be initialised only once, at the point where its first meaningful value is determined. This approach will improve code clarity and maintainability, ensuring that unnecessary operations are avoided and the code intent is more explicit.

SAE - STL Algorithm Efficiency

Criticality	Minor / Informative
Location	Wallet/WalletGreen.cpp:1772,2336,2471,2776,2889,2904
Status	Unresolved

Description

The code contains several instances where raw loops are used, which could be replaced with more efficient and readable Standard Template Library (STL) algorithms. In Wallet/WalletGreen.cpp, there are opportunities to use algorithms such as `std::find_if` for searches and `std::transform` for transformations instead of raw loops. For example, the address validation, emplacing back elements, and pushing back amounts or transfers can all be rewritten using STL algorithms. Leveraging these algorithms would enhance readability, simplify the code, and in some cases improve performance due to their optimised implementations.

Recommendation

To improve efficiency, readability, and maintainability, it is recommended to refactor the identified raw loops to use STL algorithms such as `std::find_if` and `std::transform` where applicable. These algorithms provide a clearer and more expressive way to achieve the same functionality, adhering to modern C++ practices. Updating the affected areas will result in more concise and maintainable code while taking advantage of the optimisations offered by STL.

UMV - Uninitialized Member Variable

Criticality	Minor / Informative
Location	Wallet/WalletSerializationV1.cpp:120 WalletLegacy/WalletUnconfirmedTransactions.h:73
Status	Unresolved

Description

The code contains instances where member variables are not properly initialised in constructors. In WalletSerializationV1.cpp, the variables `WalletTransferDto::amount` and `WalletTransferDto::type` are left uninitialised in the constructor, potentially leading to undefined behaviour if accessed before being explicitly assigned a value. Similarly, in WalletUnconfirmedTransactions.h, the variable `UnconfirmedTransferDetails::outsAmount` is not set during construction. These omissions can result in unpredictable program outcomes and obscure potential bugs.

Recommendation

It is recommended to initialise all member variables in constructors to ensure predictable and robust behaviour. This can be achieved by using an initialisation list in the constructor or setting default values within the class definition. Properly initialising member variables ensures a well-defined state for objects from the moment they are created, reducing the likelihood of bugs and improving the reliability and maintainability of the code.

UAV - Unutilized Assigned Variable

Criticality	Minor / Informative
Location	Wallet/WalletGreen.cpp:784
Status	Unresolved

Description

The code contains an instance where a variable is assigned a value but is never used. In Wallet/WalletGreen.cpp, the variable `updatedTransactions` is assigned the result of `deleteTransfersForAddress(deletedAddressString, deletedTransactions)` but is not utilised anywhere in the code. This can lead to unnecessary resource consumption and adds clutter, potentially obscuring the code's intent. Such unutilized variables can confuse developers and mask areas where logical improvements or simplifications may be needed.

Recommendation

It is recommended to review these variables and either remove them if they serve no purpose or revise the code to ensure their values are utilized. Doing so will improve resource efficiency, reduce code clutter, and enhance maintainability and clarity.

Summary

The Dynex code implements a comprehensive digital wallet system in the release. This audit investigates security issues, business logic concerns, potential improvements in performance, and adherence to best coding practices. The code review and auditing process have uncovered some key areas for improvement within the C++ codebase.

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