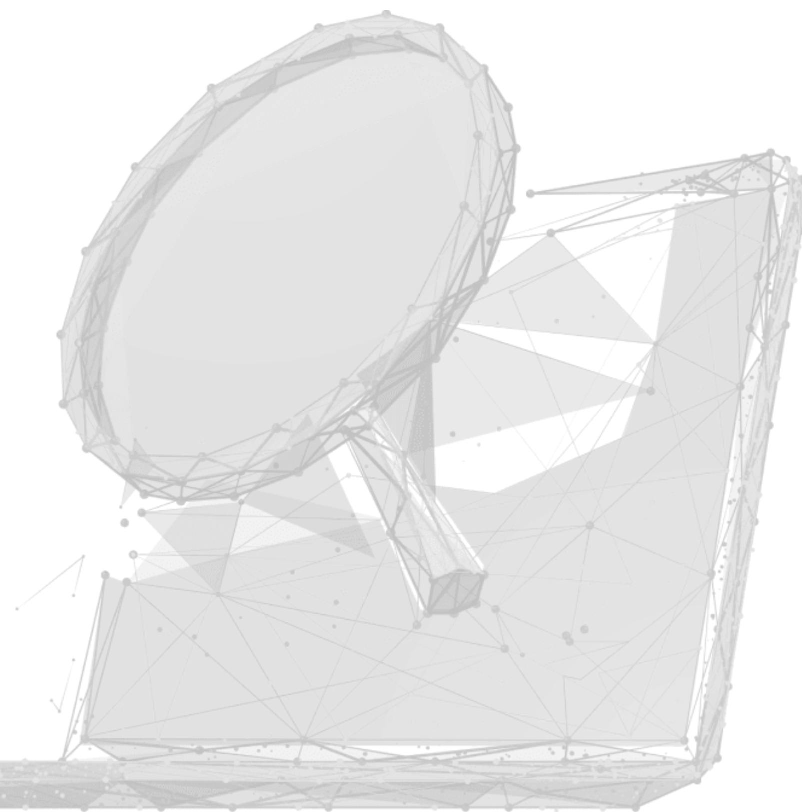
BLOCKCHAIN NETWORK AUDIT

NEXIS NETWORK



Prepared by:
The Datami team
account@datami.ee

CONTENTS

OUR COMPANY

Datami is a distinguished team of highly qualified and experienced white hat hackers. With a rich market presence of over 7 years, we have established our offices in locations including Chernivtsi, Ukraine; Tallinn, Estonia; and New York, USA. Throughout our journey since 2015, we have successfully collaborated with over 600 clients from diverse countries such as the USA, Germany, Canada, Australia, UAE, Poland, Japan, China, Latvia, Estonia, Cayman Islands, Cyprus, and many more. Our expertise extends internationally.

We have worked with entities such as banks, cryptocurrency exchanges, e-commerce platforms, the US defense industry, fintech organizations, streaming platforms, news resources, and more.

Our team is proficient in handling a diverse range of tasks:

1) Carry out regular penetration tests or IT audits: networks, operating systems, services, software, Wi-Fi, databases, mobile applications (reverse engineering of Android, iOS apps), and Web-applications;
2) Provide recommendations and consultations for eliminating the identified vulnerabilities in accordance with the results of pentests;
3) Implement our proprietary solution to protect your resource 24/7 to mitigating concerns arising from external threats.
4) Act as an ambulance in cases of hacking or DDOS-attacks;
5) Provide any kind of solution in Cybersecurity for our clients.

It's our goal to give our clients a service they can rely on. You can trust us to protect your digital assets and keep your organization safe from evolving cyber threats with our profound expertise, dedication, and commitment to excellence.

NEXIS NETWORK DESCRIPTION

Nexis-Network is a decentralized layer-1 blockchain designed to support AI data analytics and real-world assets (RWA). It offers a high-speed, scalable, and low-cost infrastructure for decentralized applications, featuring parallel processing for real-time transaction finalization and horizontal scaling to keep fees low. The network is EVM and eBFT interoperable, supports WebAssembly (WASM) for high-performance smart contracts, and ensures carbon neutrality through an energy-efficient DPoS mechanism. Nexis aims to power the future of decentralized AI and computing, making it accessible and efficient for developers and enterprises alike.

DISCLAIMER

This report includes our findings based on the analysis of security vulnerabilities and issues in the Rust-based Nexis-Network project static source code, following good industry practices as of the report date. To fully understand the scope and details of the findings, it is essential to read the entire report. While every effort was made to ensure thorough analysis and accurate reporting, this document should not be solely relied upon for security assurance. We recommend additional audits by other teams and a bug bounty program on a test network before deploying the network to a production environment. These recommendations are based on effective security strategies.

This report is not:

- A guarantee of the security of the blockchain network.
- A guarantee of future resilience against attacks against the network . This includes unknown and undocumented attacks at the time of the audit.
- An absolute determinant of all security issues that may exist within an audited project.
- A guarantee of the security of the network, if the blockchain is compromised through the owner account, or otherwise designated high permissioned account, maliciously using the project, whether through a rogue actor or the administrative accounts being compromised.

EXECUTIVE SUMMARY

Blockchain networks are the foundation of decentralized applications and token ecosystems on Nexis-Network. Ensuring the security of these networks involves rigorous audits to detect vulnerabilities and logic flaws that could be exploited by attackers. These audits are critical for maintaining trust and reliability, given the immutable nature of blockchain transactions and the high-value assets they manage. The process involves a combination of automated and manual techniques to evaluate the network's defenses against potential threats, ensuring compliance with best practices in blockchain security.

Audits focus on identifying potential compromises, sensitive information theft, and weak conditions that could lead to fraud or security incidents. By using tools and techniques similar to those an attacker might employ, auditors can thoroughly verify and secure the network. The goal is to safeguard the network against any threats, ensuring it meets the highest security standards and protects the interests of users and stakeholders.

PROJECT APPROACH

Service agreement

Prior to the engagement, Customer and Executor established a service agreement. This type of contract lists the services that the provider will perform and details the time frame and compensation for the project. The service contract also lists the rights and requirements of both parties, including liabilities and confidentiality guidelines, as well as a dispute resolution in case either party breaches the contract.

These rules provide permission to conduct testing and outline the procedures for notification of vulnerability scanning, notification of vulnerabilities and vulnerability exploitation.

FINDINGS AND RECOMMENDATIONS KEY

Wherever possible, security audit rate each finding in this document according to its business impact and each recommendation in terms of the effort required in correcting the problem. The following table describes the different rating levels.

Risk type	**This column provides a brief technical description of the finding.**

CRITICAL	These vulnerabilities must be processed instantly due to the high grade of threat they show to the network, users or critical infrastructure. For this kind of vulnerability, user does not require advanced tools or special techniques or advanced knowledge.
HIGH	These vulnerabilities must be processed instantly due to the high grade of threat they show for the network, users or data. These vulnerabilities don't require a skilled attacker that possesses advanced tools in order to be exploited, therefore they need to be addressed as soon as possible. Could result in a loss of funds for the contract owner or users.
MEDIUM	This vulnerability class needs to be addressed in time. Exploitation is commonly tough and requires social engineering, existing access or special circumstances Results in the code specification operating incorrectly.
LOW	These vulnerabilities should be taken into consideration and possessed in the future. These issues offer limited information possibilities to an invader and may not be a real threat. A best practice or design issue that could affect the security standard of the contract.
INFO	These are informational disclosure and have very low chances to be used as a real threat. The issue addresses a violation in best practice or a design pattern that has a minimal risk of affecting the security of the contract.

SUMMARY OF FINDINGS

The audit was conducted on 10 May to 3 June 2024 against the blockchain network provided by the Nexis company. The assessment was conducted in a manner that simulated a malicious individual who has access to the Nexis company project source code. The objective was to verify the target object's security and determine whether an attacker could compromise the target object’s defense.

The goal of the audit for the scoped part of the network was met. It was determined that it is **not possible to misuse the network functionality in the audited part of the project**.

Vulnerabilities	Risk level	Confidence	Severity
Multiple calls are executed in the same transaction	INFO	INFO	INFO

The nature of information threats considers the uncertainty of ways to misuse web3 project, that may be used by an attacker. In addition, the set of known technical vulnerabilities of the libraries, components, and hosting environment is constantly increasing. Therefore, the results of this audit cannot guarantee to uncover all possible compromise or penetration ways and security problems, and only show the weakest points in the security of the target object.

Besides audits, to enhance Customer security effectively and to reduce Customer business risks, other appropriate security management processes and security solutions should be designed and implemented. These security measures include but are not limited to a secure development lifecycle, regular security audits by an independent party, security event monitoring, and incident response.

PROJECT APPROACH

Rules of engagement

Prior to the engagement, auditors established the rules of the engagement for the assessment. These rules provided the permission to conduct testing and outlined the procedures for notification of vulnerability scanning, notification of vulnerabilities, and vulnerability exploitation.

Blockchain audit methodology

The security audit was done using a combination of manual and automated tools and techniques to identify vulnerabilities within the target environment and exploit them.

Phase 1 – Getting familiar with the code and application

Following steps were carried out during this security audit.

Step	Description
Brief code overview	To quickly get familiar with project functionality, unit tests were executed and graph images were generated and reviewed. Particular attention was paid to analyze cryptography, third-party modules, and the structure of libraries.

Phase 2 – Project source code analysis

Within this phase the team scanned the application with appropriate binary and source level to identify potential violations from coding guidelines and security practices.
The following utilities were used for project analysis:

Name	Description
rust cargo	Rust's Cargo is a package manager and build system that manages dependencies, compiles code, runs tests, and builds executables for Rust projects
cargo-audit	Cargo-audit is a tool for auditing Rust project dependencies to identify vulnerabilities, ensure code security, and check for outdated or insecure dependencies.
rust-analyzer	Rust Analyzer is an integrated development environment (IDE) tool that provides features like code completion, navigation, and real-time feedback to enhance Rust programming productivity.
graphviz	Graphviz is a graph visualization software that provides tools to create diagrams of abstract graphs and networks using a simple plain text language.
hongfuzz	Honggfuzz is an efficient fuzzing tool for testing Rust code to find security vulnerabilities and bugs.

Phase 3 – Scanners results verification

Within this phase the team reviews scan results to identify which of them are false positives and which of them can affect application security.

PROJECT ANALYSIS

Tokens for named pipes may be delivered after deregistration

Risk level: CLOSED Confidence: CLOSED Severity: CLOSED

Description:

When an I/O resource is registered with Mio, a readiness event is delivered to the user using a specified token. Mio ensures that once an I/O resource is deregistered, the associated token will no longer be returned. However, on Windows, there is a vulnerability with named pipes. In some instances, Mio may incorrectly deliver the token for a named pipe even after it has been deregistered. This inconsistency can lead to potential security risks or unintended behavior in applications relying on Mio for I/O event handling.

Recommendations:

Update the mio package to version >=0.8.11 to address the vulnerability.

```
root@WinDev2404Eval:/mnt/c/Users/User/Desktop/folder/NexisAudit/nexis-network# grep -r -w -I 'Pipe' .
./logger/src/lib.rs:        .target(env_logger::Target::Pipe(Box::new(file)))
```

```
root@WinDev2404Eval:/mnt/c/Users/User/Desktop/folder/NexisAudit/nexis-network# cargo search mio
mio = "0.8.11"              # Lightweight non-blocking I/O.
```

Stack overflow in rustc_serialize when parsing deeply nested JSON

Risk level: CLOSED Confidence: CLOSED Severity: CLOSED

Description:

When parsing JSON using json::Json::from_str, there is no limit to the depth of the stack, therefore deeply nested objects can cause a stack overflow, which aborts the process.

Recommendations:

Replace rustc-serialize with Serde for safe JSON parsing.

```
root@WinDev2404Eval:/mnt/c/Users/User/Desktop/folder/NexisAudit/nexis-network# cargo search rustc-serialize
rustc-serialize = "0.3.25"                # Generic serialization/deserialization support corresponding to the `derive(Rust…
core_rustc-serialize = "0.3.20-v0.3.19patch1"   # Generic serialization/deserialization support corresponding to the `derive(Rust…
json_macros = "0.3.2"                     # Convenience macros for constructing JSON objects from literals.
rustc-serialize2 = "0.3.25"               # Generic serialization/deserialization support corresponding to the `derive(Rust…
```

Multiple issues involving quote API

Risk level: **CLOSED** Confidence: **CLOSED** Severity: **CLOSED**

Description:

A vulnerability has been identified in the shlex crate (version 1.3.0) used in Rust projects. This crate is intended to split strings into shell words, similar to Python's shlex library.

Recommendations:

Update shlex to version >=1.3.0 to address vulnerabilities related to the quote API.

```
root@WinDev2404Eval:/mnt/c/Users/User/Desktop/folder/NexisAudit/nexis-network# cargo search shlex
shlex = "1.3.0"                  # Split a string into shell words, like Python's shlex.
```

rustls::ConnectionCommon::complete_io could fall into an infinite loop based on network input

Risk level: **CLOSED** Confidence: **CLOSED** Severity: **CLOSED**

Description:

If a close_notify alert is received during a handshake, the complete_io function does not terminate as expected. This issue only affects callers that invoke the complete_io function. Specifically, rustls-tokio and rustls-ffi do not call complete_io and are therefore not affected by this vulnerability. However, the rustls::Stream and rustls::StreamOwned types utilize complete_io and are impacted by this issue.

Recommendations:

Update the rustls package to version >=0.23.5, >=0.22.4, <0.23.0, or >=0.21.11, <0.22.0 to address the vulnerability

```
./client/src/nonblocking/quic_client.rs:impl rustls::client::ServerCertVerifier for SkipServerVerification {
./client/src/nonblocking/quic_client.rs:        _end_entity: &rustls::Certificate,
./client/src/nonblocking/quic_client.rs:        _intermediates: &[rustls::Certificate],
./client/src/nonblocking/quic_client.rs:        _server_name: &rustls::ServerName,
./client/src/nonblocking/quic_client.rs:    ) -> Result<rustls::client::ServerCertVerified, rustls::Error> {
./client/src/nonblocking/quic_client.rs:        Ok(rustls::client::ServerCertVerified::assertion())
./client/src/nonblocking/quic_client.rs:    pub certificates: Vec<rustls::Certificate>,
./client/src/nonblocking/quic_client.rs:    pub key: rustls::PrivateKey,
./client/src/nonblocking/quic_client.rs:        let mut crypto = rustls::ClientConfig::builder()
```

```
./streamer/src/nonblocking/quic.rs:    impl rustls::client::ServerCertVerifier for SkipServerVerification {
./streamer/src/nonblocking/quic.rs:            _end_entity: &rustls::Certificate,
./streamer/src/nonblocking/quic.rs:            _intermediates: &[rustls::Certificate],
./streamer/src/nonblocking/quic.rs:            _server_name: &rustls::ServerName,
./streamer/src/nonblocking/quic.rs:        ) -> Result<rustls::client::ServerCertVerified, rustls::Error> {
./streamer/src/nonblocking/quic.rs:            Ok(rustls::client::ServerCertVerified::assertion())
./streamer/src/nonblocking/quic.rs:        let mut crypto = rustls::ClientConfig::builder()
```

Potential segfault in the time crate

Risk level: **CLOSED** Confidence: **CLOSED** Severity: **CLOSED**

Description:

On Unix-like operating systems, there is a risk of segmentation faults caused by dereferencing a dangling pointer under specific conditions. This issue arises when an environment variable is set in a different thread than the one executing the affected functions. The problem can occur without the user's knowledge, especially if it is triggered by a third-party library.

Recommendations:

Update time to version >=0.2.23 or version 0.3.x which is not vulnerable.

tokio: reject_remote_clients Configuration corruption

Risk level: **CLOSED** Confidence: **CLOSED** Severity: **CLOSED**

Description:

On Windows, configuring a named pipe server with pipe_mode will inadvertently set ServerOptions::reject_remote_clients to false. This means that the server will not reject remote clients, potentially exposing the server to unauthorized access from remote sources.

Recommendations:

Update tokio to version >=1.23.1 to fix the reject_remote_clients configuration vulnerability.

```
root@WinDev2404Eval:/mnt/c/Users/User/Desktop/folder/NexisAudit/nexis-network# cargo search tokio
tokio = "1.38.0"                  # An event-driven, non-blocking I/O platform for writing asynchronous I/O backed applications…
```

webpki: CPU denial of service in certificate path building

Risk level: **CLOSED** Confidence: **CLOSED** Severity: **CLOSED**

Description:

When this crate is provided with a pathological certificate chain for validation, it will consume an exponential amount of CPU time relative to the number of candidate certificates at each step of the path-building process. This can lead to significant performance degradation and potential denial of service due to excessive CPU usage.

Recommendations:

Update webpki to version >=0.22.2 to address the vulnerability.

```
root@WinDev2404Eval:/mnt/c/Users/User/Desktop/folder/NexisAudit/nexis-network# cargo search webpki
webpki = "0.22.4"                  # Web PKI X.509 Certificate Verification.
```

CODE REVIEW

A thorough security code review was conducted on the Nexis Network source code, focusing on potential vulnerabilities listed in the checklist. Auditors meticulously analyzed the code to identify any possible attack vectors, including logic flaws, integer overflows, race conditions, and improper access controls. The review ensured that the code adhered to industry security standards, highlighting areas that were susceptible to attacks and suggesting necessary mitigations.

parse_account_data.rs

```
pub fn parse_account_data(
    pubkey: &Pubkey,
    program_id: &Pubkey,
    data: &[u8],
    additional_data: Option<AccountAdditionalData>,
) -> Result<ParsedAccount, ParseAccountError> {
    let program_name = PARSABLE_PROGRAM_IDS
        .get(program_id)
        .ok_or(ParseAccountError::ProgramNotParsable)?;
    let additional_data = additional_data.unwrap_or_default();
    let parsed_json = match program_name {
        ParsableAccount::BpfUpgradeableLoader => {
            serde_json::to_value(parse_bpf_upgradeable_loader(data)?)?
        }
        ParsableAccount::Config => serde_json::to_value(parse_config(data, pubkey)?)?,
        ParsableAccount::Nonce => serde_json::to_value(parse_nonce(data)?)?,
        ParsableAccount::SplToken | ParsableAccount::SplToken2022 => {
            serde_json::to_value(parse_token(data, additional_data.spl_token_decimals)?)?
        }
        ParsableAccount::Stake => serde_json::to_value(parse_stake(data)?)?,
        ParsableAccount::Sysvar => serde_json::to_value(parse_sysvar(data, pubkey)?)?,
        ParsableAccount::Vote => serde_json::to_value(parse_vote(data)?)?,
        ParsableAccount::NexisAccount => {
            serde_json::to_value(nexis_account_program::NexisAccountType::try_from(data)?)?
        }
        ParsableAccount::NexisRelyingParty => serde_json::to_value(
            nexis_relying_party_program::RelyingPartyData::try_from(data)?,
        )?,
    };
    Ok(ParsedAccount {
        program: format!("{:?}", program_name).to_kebab_case(),
        parsed: parsed_json,
        space: data.len() as u64,
    })
}
```

Logic flaws (Unpredictable states)	OK
Integer overflows	OK
Uninitialized variables	OK
Improper secrets handling	OK
Type confusions	OK
Denial of service issues	OK
Race conditions	OK
Improper access controls	OK

Unsafe code usage	OK
Third-party libraries and dependencies review	OK
Exception disorders	OK
Unchecked return values	OK
Outdated compiler version	OK
Implicit visibility	OK
Weak randomness sources	OK
Short address issues	OK
Denial of Service (DoS) through resource exhaustion	OK
Side-channel attacks	OK
Data serialization and deserialization flaws	OK
Memory safety	OK
Cryptographic weaknesses	OK
Concurrency issues	OK
Network security	OK
Logging and error handling	OK

parse_bpf_loader.rs

```rust
pub fn parse_bpf_upgradeable_loader(
    data: &[u8],
) -> Result<BpfUpgradeableLoaderAccountType, ParseAccountError> {
    let account_state: UpgradeableLoaderState = deserialize(data).map_err(|_| {
        ParseAccountError::AccountNotParsable(ParsableAccount::BpfUpgradeableLoader)
    })?;
    let parsed_account = match account_state {
        UpgradeableLoaderState::Uninitialized => BpfUpgradeableLoaderAccountType::Uninitialized,
        UpgradeableLoaderState::Buffer { authority_address } => {
            let offset = if authority_address.is_some() {
                UpgradeableLoaderState::buffer_data_offset().unwrap()
            } else {
                // This case included for code completeness; in practice, a Buffer account will
                // always have authority_address.is_some()
                UpgradeableLoaderState::buffer_data_offset().unwrap()
                    - serialized_size(&Pubkey::default()).unwrap() as usize
            };
            BpfUpgradeableLoaderAccountType::Buffer(UiBuffer {
                authority: authority_address.map(|pubkey| pubkey.to_string()),
                data: UiAccountData::Binary(
                    base64::encode(&data[offset..]),
                    UiAccountEncoding::Base64,
                ),
            })
        }
        UpgradeableLoaderState::Program {
            programdata_address,
        } => BpfUpgradeableLoaderAccountType::Program(UiProgram {
            program_data: programdata_address.to_string(),
        }),
        UpgradeableLoaderState::ProgramData {
            slot,
            upgrade_authority_address,
        } => {
            let offset = if upgrade_authority_address.is_some() {
                UpgradeableLoaderState::programdata_data_offset().unwrap()
            } else {
                UpgradeableLoaderState::programdata_data_offset().unwrap()
                    - serialized_size(&Pubkey::default()).unwrap() as usize
            };
            BpfUpgradeableLoaderAccountType::ProgramData(UiProgramData {
                slot,
                authority: upgrade_authority_address.map(|pubkey| pubkey.to_string()),
                data: UiAccountData::Binary(
                    base64::encode(&data[offset..]),
                    UiAccountEncoding::Base64,
                ),
            })
        }
    };
    Ok(parsed_account)
}
```

Logic flaws (Unpredictable states)	OK
Integer overflows	OK
Uninitialized variables	OK
Improper secrets handling	OK
Type confusions	OK
Denial of service issues	OK
Race conditions	OK
Improper access controls	OK
Unsafe code usage	OK
Third-party libraries and dependencies review	OK
Exception disorders	OK
Unchecked return values	OK
Outdated compiler version	OK
Implicit visibility	OK
Weak randomness sources	OK
Short address issues	OK
Denial of Service (DoS) through resource exhaustion	OK
Side-channel attacks	OK
Data serialization and deserialization flaws	OK
Memory safety	OK
Cryptographic weaknesses	OK
Concurrency issues	OK
Network security	OK
Logging and error handling	OK

parse_config.rs

```rust
pub fn parse_config(data: &[u8], pubkey: &Pubkey) -> Result<ConfigAccountType, ParseAccountError> {
    let parsed_account = if pubkey == &stake_config::id() {
        get_config_data(data)
            .ok()
            .and_then(|data| deserialize::<StakeConfig>(data).ok())
            .map(|config| ConfigAccountType::StakeConfig(config.into()))
    } else {
        deserialize::<ConfigKeys>(data).ok().and_then(|key_list| {
            if !key_list.keys.is_empty() && key_list.keys[0].0 == validator_info::id() {
                parse_config_data::<String>(data, key_list.keys).and_then(|validator_info| {
                    Some(ConfigAccountType::ValidatorInfo(UiConfig {
                        keys: validator_info.keys,
                        config_data: serde_json::from_str(&validator_info.config_data).ok()?,
                    }))
                })
            } else {
                None
            }
        })
    };
    parsed_account.ok_or(ParseAccountError::AccountNotParsable(
        ParsableAccount::Config,
    ))
}
```

```rust
fn parse_config_data<T>(data: &[u8], keys: Vec<(Pubkey, bool)>) -> Option<UiConfig<T>>
where
    T: serde::de::DeserializeOwned,
{
    let config_data: T = deserialize(get_config_data(data).ok()?).ok()?;
    let keys = keys
        .iter()
        .map(|key| UiConfigKey {
            pubkey: key.0.to_string(),
            signer: key.1,
        })
        .collect();
    Some(UiConfig { keys, config_data })
}
```

Logic flaws (Unpredictable states)	OK
Integer overflows	OK
Uninitialized variables	OK
Improper secrets handling	OK
Type confusions	OK
Denial of service issues	OK
Race conditions	OK
Improper access controls	OK

Unsafe code usage	OK
Third-party libraries and dependencies review	OK
Exception disorders	OK
Unchecked return values	OK
Outdated compiler version	OK
Implicit visibility	OK
Weak randomness sources	OK
Short address issues	OK
Denial of Service (DoS) through resource exhaustion	OK
Side-channel attacks	OK
Data serialization and deserialization flaws	OK
Memory safety	OK
Cryptographic weaknesses	OK
Concurrency issues	OK
Network security	OK
Logging and error handling	OK

parse_nonce.rs

```rust
pub fn parse_nonce(data: &[u8]) -> Result<UiNonceState, ParseAccountError> {
    let nonce_versions: Versions = bincode::deserialize(data)
        .map_err(|_| ParseAccountError::from(InstructionError::InvalidAccountData))?;
    match nonce_versions.state() {
        // This prevents parsing an allocated System-owned account with empty data of any non-zero
        // length as `uninitialized` nonce. An empty account of the wrong length can never be
        // initialized as a nonce account, and an empty account of the correct length may not be an
        // uninitialized nonce account, since it can be assigned to another program.
        State::Uninitialized => Err(ParseAccountError::from(
            InstructionError::InvalidAccountData,
        )),
        State::Initialized(data) => Ok(UiNonceState::Initialized(UiNonceData {
            authority: data.authority.to_string(),
            blockhash: data.blockhash().to_string(),
            fee_calculator: data.fee_calculator.into(),
        })),
    }
}
```

Logic flaws (Unpredictable states)	OK
Integer overflows	OK
Uninitialized variables	OK
Improper secrets handling	OK
Type confusions	OK
Denial of service issues	OK
Race conditions	OK
Improper access controls	OK
Unsafe code usage	OK
Third-party libraries and dependencies review	OK
Exception disorders	OK
Unchecked return values	OK
Outdated compiler version	OK
Implicit visibility	OK
Weak randomness sources	OK
Short address issues	OK
Denial of Service (DoS) through resource exhaustion	OK
Side-channel attacks	OK
Data serialization and deserialization flaws	OK
Memory safety	OK
Cryptographic weaknesses	OK

Concurrency issues	OK
Network security	OK
Logging and error handling	OK

parse_stake.rs

```rust
pub fn parse_stake(data: &[u8]) -> Result<StakeAccountType, ParseAccountError> {
    let stake_state: StakeState = deserialize(data)
        .map_err(|_| ParseAccountError::AccountNotParsable(ParsableAccount::Stake))?;
    let parsed_account = match stake_state {
        StakeState::Uninitialized => StakeAccountType::Uninitialized,
        StakeState::Initialized(meta) => StakeAccountType::Initialized(UiStakeAccount {
            meta: meta.into(),
            stake: None,
        }),
        StakeState::Stake(meta, stake) => StakeAccountType::Delegated(UiStakeAccount {
            meta: meta.into(),
            stake: Some(stake.into()),
        }),
        StakeState::RewardsPool => StakeAccountType::RewardsPool,
    };
    Ok(parsed_account)
}
```

Logic flaws (Unpredictable states)	OK
Integer overflows	OK
Uninitialized variables	OK
Improper secrets handling	OK
Type confusions	OK
Denial of service issues	OK
Race conditions	OK
Improper access controls	OK
Unsafe code usage	OK
Third-party libraries and dependencies review	OK
Exception disorders	OK
Unchecked return values	OK
Outdated compiler version	OK
Implicit visibility	OK
Weak randomness sources	OK
Short address issues	OK
Denial of Service (DoS) through resource exhaustion	OK

Side-channel attacks	OK
Data serialization and deserialization flaws	OK
Memory safety	OK
Cryptographic weaknesses	OK
Concurrency issues	OK
Network security	OK
Logging and error handling	OK

parse_sysvar.rs

```
pub fn parse_sysvar(data: &[u8], pubkey: &Pubkey) -> Result<SysvarAccountType, ParseAccountError> {
    #[allow(deprecated)]
    let parsed_account = {
        if pubkey == &sysvar::clock::id() {
            deserialize::<Clock>(data)
                .ok()
                .map(|clock| SysvarAccountType::Clock(clock.into()))
        } else if pubkey == &sysvar::epoch_schedule::id() {
            deserialize(data).ok().map(SysvarAccountType::EpochSchedule)
        } else if pubkey == &sysvar::fees::id() {
            deserialize::<Fees>(data)
                .ok()
                .map(|fees| SysvarAccountType::Fees(fees.into()))
        } else if pubkey == &sysvar::recent_blockhashes::id() {
            deserialize::<RecentBlockhashes>(data)
                .ok()
                .map(|recent_blockhashes| {
                    let recent_blockhashes = recent_blockhashes
                        .iter()
                        .map(|entry| UiRecentBlockhashesEntry {
                            blockhash: entry.blockhash.to_string(),
                            fee_calculator: entry.fee_calculator.into(),
                        })
                        .collect();
                    SysvarAccountType::RecentBlockhashes(recent_blockhashes)
                })
        } else if pubkey == &sysvar::rent::id() {
            deserialize::<Rent>(data)
                .ok()
                .map(|rent| SysvarAccountType::Rent(rent.into()))
        } else if pubkey == &sysvar::rewards::id() {
            deserialize::<Rewards>(data)
                .ok()
                .map(|rewards| SysvarAccountType::Rewards(rewards.into()))
        } else if pubkey == &sysvar::slot_hashes::id() {
            deserialize::<SlotHashes>(data).ok().map(|slot_hashes| {
                let slot_hashes = slot_hashes
```

```
                    .iter()
                    .map(|slot_hash| UiSlotHashEntry {
                        slot: slot_hash.0,
                        hash: slot_hash.1.to_string(),
                    })
                    .collect();
                SysvarAccountType::SlotHashes(slot_hashes)
            })
        } else if pubkey == &sysvar::slot_history::id() {
            deserialize::<SlotHistory>(data).ok().map(|slot_history| {
                SysvarAccountType::SlotHistory(UiSlotHistory {
                    next_slot: slot_history.next_slot,
                    bits: format!("{:?}", SlotHistoryBits(slot_history.bits)),
                })
            })
        } else if pubkey == &sysvar::stake_history::id() {
            deserialize::<StakeHistory>(data).ok().map(|stake_history| {
                let stake_history = stake_history
                    .iter()
                    .map(|entry| UiStakeHistoryEntry {
                        epoch: entry.0,
                        stake_history: entry.1.clone(),
                    })
                    .collect();
                SysvarAccountType::StakeHistory(stake_history)
            })
        } else {
            None
        }
    };
    parsed_account.ok_or(ParseAccountError::AccountNotParsable(
        ParsableAccount::Sysvar,
    ))
}
```

Logic flaws (Unpredictable states)	OK
Integer overflows	OK
Uninitialized variables	OK
Improper secrets handling	OK
Type confusions	OK
Denial of service issues	OK
Race conditions	OK
Improper access controls	OK
Unsafe code usage	OK

Third-party libraries and dependencies review	OK
Exception disorders	OK
Unchecked return values	OK
Outdated compiler version	OK
Implicit visibility	OK
Weak randomness sources	OK
Short address issues	OK
Denial of Service (DoS) through resource exhaustion	OK
Side-channel attacks	OK
Data serialization and deserialization flaws	OK
Memory safety	OK
Cryptographic weaknesses	OK
Concurrency issues	OK
Network security	OK
Logging and error handling	OK

parse_token_extension.rs

```
            .get_extension::<extension::interest_bearing_mint::InterestBearingConfig>()
            .map(|&extension| UiExtension::InterestBearingConfig(extension.into()))
            .unwrap_or(UiExtension::UnparseableExtension),
    }
}
```

```rust
pub fn parse_extension<S: BaseState>(
    extension_type: &ExtensionType,
    account: &StateWithExtensions<S>,
) -> UiExtension {
    match &extension_type {
        ExtensionType::Uninitialized => UiExtension::Uninitialized,
        ExtensionType::TransferFeeConfig => account
            .get_extension::<extension::transfer_fee::TransferFeeConfig>()
            .map(|&extension| UiExtension::TransferFeeConfig(extension.into()))
            .unwrap_or(UiExtension::UnparseableExtension),
        ExtensionType::TransferFeeAmount => account
            .get_extension::<extension::transfer_fee::TransferFeeAmount>()
            .map(|&extension| UiExtension::TransferFeeAmount(extension.into()))
            .unwrap_or(UiExtension::UnparseableExtension),
        ExtensionType::MintCloseAuthority => account
            .get_extension::<extension::mint_close_authority::MintCloseAuthority>()
            .map(|&extension| UiExtension::MintCloseAuthority(extension.into()))
            .unwrap_or(UiExtension::UnparseableExtension),
        ExtensionType::ConfidentialTransferMint => account
            .get_extension::<extension::confidential_transfer::ConfidentialTransferMint>()
            .map(|&extension| UiExtension::ConfidentialTransferMint(extension.into()))
            .unwrap_or(UiExtension::UnparseableExtension),
        ExtensionType::ConfidentialTransferAccount => account
            .get_extension::<extension::confidential_transfer::ConfidentialTransferAccount>()
            .map(|&extension| UiExtension::ConfidentialTransferAccount(extension.into()))
            .unwrap_or(UiExtension::UnparseableExtension),
        ExtensionType::DefaultAccountState => account
            .get_extension::<extension::default_account_state::DefaultAccountState>()
            .map(|&extension| UiExtension::DefaultAccountState(extension.into()))
            .unwrap_or(UiExtension::UnparseableExtension),
        ExtensionType::ImmutableOwner => UiExtension::ImmutableOwner,
        ExtensionType::MemoTransfer => account
            .get_extension::<extension::memo_transfer::MemoTransfer>()
            .map(|&extension| UiExtension::MemoTransfer(extension.into()))
            .unwrap_or(UiExtension::UnparseableExtension),
        ExtensionType::NonTransferable => UiExtension::NonTransferable,
        ExtensionType::InterestBearingConfig => account
```

```rust
    fn from(
        confidential_transfer_account: extension::confidential_transfer::ConfidentialTransferAccount,
    ) -> Self {
        Self {
            approved: confidential_transfer_account.approved.into(),
            encryption_pubkey: format!("{}", confidential_transfer_account.encryption_pubkey),
            pending_balance_lo: format!("{}", confidential_transfer_account.pending_balance_lo),
            pending_balance_hi: format!("{}", confidential_transfer_account.pending_balance_hi),
            available_balance: format!("{}", confidential_transfer_account.available_balance),
            decryptable_available_balance: format!(
                "{}",
                confidential_transfer_account.decryptable_available_balance
            ),
            allow_balance_credits: confidential_transfer_account.allow_balance_credits.into(),
            pending_balance_credit_counter: confidential_transfer_account
                .pending_balance_credit_counter
                .into(),
            maximum_pending_balance_credit_counter: confidential_transfer_account
                .maximum_pending_balance_credit_counter
                .into(),
            expected_pending_balance_credit_counter: confidential_transfer_account
                .expected_pending_balance_credit_counter
                .into(),
            actual_pending_balance_credit_counter: confidential_transfer_account
                .actual_pending_balance_credit_counter
                .into(),
            withheld_amount: format!("{}", confidential_transfer_account.withheld_amount),
        }
    }
}
```

Logic flaws (Unpredictable states)	OK

Integer overflows	OK
Uninitialized variables	OK
Improper secrets handling	OK
Type confusions	OK
Denial of service issues	OK
Race conditions	OK
Improper access controls	OK
Unsafe code usage	OK
Third-party libraries and dependencies review	OK
Exception disorders	OK
Unchecked return values	OK
Outdated compiler version	OK
Implicit visibility	OK
Weak randomness sources	OK
Short address issues	OK
Denial of Service (DoS) through resource exhaustion	OK
Side-channel attacks	OK
Data serialization and deserialization flaws	OK
Memory safety	OK
Cryptographic weaknesses	OK
Concurrency issues	OK
Network security	OK
Logging and error handling	OK

parse_token.rs

```
pub(crate) fn spl_token_id() -> Pubkey {
    Pubkey::new_from_array(spl_token::id().to_bytes())
}
```

```
pub(crate) fn spl_token_2022_id() -> Pubkey {
    Pubkey::new_from_array(spl_token_2022::id().to_bytes())
}
```

```
pub fn spl_token_ids() -> Vec<Pubkey> {
    vec![spl_token_id(), spl_token_2022_id()]
}
```

```rust
pub fn is_known_spl_token_id(program_id: &Pubkey) -> bool {
    *program_id == spl_token_id() || *program_id == spl_token_2022_id()
}
```

```rust
pub fn spl_token_native_mint() -> Pubkey {
    Pubkey::new_from_array(spl_token::native_mint::id().to_bytes())
}
```

```rust
pub fn spl_token_native_mint_program_id() -> Pubkey {
    spl_token_id()
}
```

```rust
pub fn spl_token_pubkey(pubkey: &Pubkey) -> SplTokenPubkey {
    SplTokenPubkey::new_from_array(pubkey.to_bytes())
}
```

```rust
pub fn pubkey_from_spl_token(pubkey: &SplTokenPubkey) -> Pubkey {
    Pubkey::new_from_array(pubkey.to_bytes())
}
```

```rust
pub fn parse_token(
    data: &[u8],
    mint_decimals: Option<u8>,
) -> Result<TokenAccountType, ParseAccountError> {
    if let Ok(account) = StateWithExtensions::<Account>::unpack(data) {
        let decimals = mint_decimals.ok_or_else(|| {
            ParseAccountError::AdditionalDataMissing(
                "no mint_decimals provided to parse spl-token account".to_string(),
            )
        })?;
        let extension_types = account.get_extension_types().unwrap_or_default();
        let ui_extensions = extension_types
            .iter()
            .map(|extension_type| parse_extension::<Account>(extension_type, &account))
            .collect();
        return Ok(TokenAccountType::Account(UiTokenAccount {
            mint: account.base.mint.to_string(),
            owner: account.base.owner.to_string(),
            token_amount: token_amount_to_ui_amount(account.base.amount, decimals),
            delegate: match account.base.delegate {
                COption::Some(pubkey) => Some(pubkey.to_string()),
                COption::None => None,
            },
            state: account.base.state.into(),
            is_native: account.base.is_native(),
            rent_exempt_reserve: match account.base.is_native {
                COption::Some(reserve) => Some(token_amount_to_ui_amount(reserve, decimals)),
                COption::None => None,
            },
            delegated_amount: if account.base.delegate.is_none() {
                None
            } else {
                Some(token_amount_to_ui_amount(
                    account.base.delegated_amount,
                    decimals,
                ))
            },
```

```rust
            close_authority: match account.base.close_authority {
                COption::Some(pubkey) => Some(pubkey.to_string()),
                COption::None => None,
            },
            extensions: ui_extensions,
        }));
    }
    if let Ok(mint) = StateWithExtensions::<Mint>::unpack(data) {
        let extension_types = mint.get_extension_types().unwrap_or_default();
        let ui_extensions = extension_types
            .iter()
            .map(|extension_type| parse_extension::<Mint>(extension_type, &mint))
            .collect();
        return Ok(TokenAccountType::Mint(UiMint {
            mint_authority: match mint.base.mint_authority {
                COption::Some(pubkey) => Some(pubkey.to_string()),
                COption::None => None,
            },
            supply: mint.base.supply.to_string(),
            decimals: mint.base.decimals,
            is_initialized: mint.base.is_initialized,
            freeze_authority: match mint.base.freeze_authority {
                COption::Some(pubkey) => Some(pubkey.to_string()),
                COption::None => None,
            },
            extensions: ui_extensions,
        }));
    }
    if data.len() == Multisig::get_packed_len() {
        let multisig = Multisig::unpack(data)
            .map_err(|_| ParseAccountError::AccountNotParsable(ParsableAccount::SplToken))?;
        Ok(TokenAccountType::Multisig(UiMultisig {
            num_required_signers: multisig.m,
            num_valid_signers: multisig.n,
            is_initialized: multisig.is_initialized,
            signers: multisig
                .signers
                .iter()
                .filter_map(|pubkey| {
                    if pubkey != &SplTokenPubkey::default() {
                        Some(pubkey.to_string())
                    } else {
                        None
                    }
                })
                .collect(),
        }))
    } else {
        Err(ParseAccountError::AccountNotParsable(
            ParsableAccount::SplToken,
        ))
    }
}
```

Logic flaws (Unpredictable states)	OK
Integer overflows	OK
Uninitialized variables	OK
Improper secrets handling	OK

Type confusions	OK
Denial of service issues	OK
Race conditions	OK
Improper access controls	OK
Unsafe code usage	OK
Third-party libraries and dependencies review	OK
Exception disorders	OK
Unchecked return values	OK
Outdated compiler version	OK
Implicit visibility	OK
Weak randomness sources	OK
Short address issues	OK
Denial of Service (DoS) through resource exhaustion	OK
Side-channel attacks	OK
Data serialization and deserialization flaws	OK
Memory safety	OK
Cryptographic weaknesses	OK
Concurrency issues	OK
Network security	OK
Logging and error handling	OK

parse_vote.rs

```
pub fn parse_vote(data: &[u8]) -> Result<VoteAccountType, ParseAccountError> {
    let mut vote_state = VoteState::deserialize(data).map_err(ParseAccountError::from)?;
    let epoch_credits = vote_state
        .epoch_credits()
        .iter()
        .map(|(epoch, credits, previous_credits)| UiEpochCredits {
            epoch: *epoch,
            credits: credits.to_string(),
            previous_credits: previous_credits.to_string(),
        })
        .collect();
    let votes = vote_state
        .votes
        .iter()
        .map(|lockout| UiLockout {
            slot: lockout.slot,
            confirmation_count: lockout.confirmation_count,
        })
        .collect();
    let authorized_voters = vote_state
        .authorized_voters()
        .iter()
        .map(|(epoch, authorized_voter)| UiAuthorizedVoters {
            epoch: *epoch,
            authorized_voter: authorized_voter.to_string(),
        })
        .collect();
    let prior_voters = vote_state
        .prior_voters()
        .buf()
        .iter()
        .filter(|(pubkey, _, _)| pubkey != &Pubkey::default())
        .map(
            |(authorized_pubkey, epoch_of_last_authorized_switch, target_epoch)| UiPriorVoters {
                authorized_pubkey: authorized_pubkey.to_string(),
                epoch_of_last_authorized_switch: *epoch_of_last_authorized_switch,
                target_epoch: *target_epoch,
```

```
                },
            )
            .collect();
        Ok(VoteAccountType::Vote(UiVoteState {
            node_pubkey: vote_state.node_pubkey.to_string(),
            authorized_withdrawer: vote_state.authorized_withdrawer.to_string(),
            commission: vote_state.commission,
            votes,
            root_slot: vote_state.root_slot,
            authorized_voters,
            prior_voters,
            epoch_credits,
            last_timestamp: vote_state.last_timestamp,
        }))
    }
```

Logic flaws (Unpredictable states)	OK
Integer overflows	OK
Uninitialized variables	OK
Improper secrets handling	OK
Type confusions	OK
Denial of service issues	OK
Race conditions	OK
Improper access controls	OK
Unsafe code usage	OK
Third-party libraries and dependencies review	OK
Exception disorders	OK
Unchecked return values	OK
Outdated compiler version	OK
Implicit visibility	OK
Weak randomness sources	OK
Short address issues	OK
Denial of Service (DoS) through resource exhaustion	OK
Side-channel attacks	OK
Data serialization and deserialization flaws	OK
Memory safety	OK
Cryptographic weaknesses	OK

Concurrency issues	OK
Network security	OK
Logging and error handling	OK

CONCLUSION

Auditors conducted the blockchain network audit on the scoped part of the project.

The goal of the audit was met. It was determined that it was **not possible to misuse Nexis blockchain network or to violate the Customer's business requirements** directly within the period allocated for this particular assessment.

OUR CONTACTS

EMAIL: account@datami.ee

PHONE: +3 (726) 991-424

Telegram: @Datami